UniVerse

BASIC Programmer's Guide

Part No. 70-9025-931

NOTICE

VMARK SOFTWARE, INC., makes no warranty of any kind with regard to the material contained in this manual, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

The information contained in this manual is subject to change without notice.

This manual contains proprietary information that is protected by copyright. All rights are reserved. It may not be photocopied, reproduced, or translated, in whole or in part, without the prior express written consent of VMARK SOFTWARE, INC.

Copyright © 1988–1996 VMARK SOFTWARE, INC. All rights reserved.

Trademarks

UniVerse, UniVerse Call Interface, UniVerse Objects, UV/Net, and VMARK are trademarks of VMARK SOFTWARE, INC.

Microsoft is a registered trademark of Microsoft Corporation. Open Database Connectivity, Windows, and Windows NT are trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

All other company or product names mentioned are trademarks or registered trademarks of their respective trademark holders.

Printing History

First Edition (70-8025-833) for Release 8.3.3, November 1995
Second Edition (70-9025-931) for Release 9.3.1, November 1996

How to Order Technical Documents

To order copies of documents or to obtain a catalog and price list, contact your local VMARK subsidiary or distributor, or call our main office at (508) 366-3888.

Customer Comments

We welcome your input. Please comment on this manual using the customer comment form provided in the back of the manual.

This manual is printed on recycled paper.

UniVerse BASIC Programmer's Guide
Table of Contents

Preface

Organization of This Manual vii
Documentation Conventions viii
UniVerse Documentation ix
Related Documentation xi

Chapter 1. Introduction to UniVerse BASIC

BASIC Terminology 1-2
Subroutines 1-3
Source Syntax 1-3
Statement Types 1-4
Statement Labels 1-5
Spaces or Tabs 1-5
Newlines and Sequential File I/O 1-5
Special Characters 1-6
Storing Programs 1-7
Editing Programs 1-7
    Editing Programs Within UniVerse 1-7
    Editing Programs Outside UniVerse 1-8
Getting Started 1-8

Chapter 2. Data Types, Variables, and Operators

Types of Data 2-1
    Character String Data 2-1
    Numeric Data 2-2
    Unknown Data: The Null Value 2-3
Constants 2-4

Variables 2-4
Array Variables 2-5
File Variables 2-9
Select List Variables 2-9
Expressions 2-9
Format Expressions 2-9
Operators 2-10
Arithmetic Operators 2-11
String Operators 2-12
Relational Operators 2-14
Pattern Matching Operators 2-16
IF Operator 2-16
Logical Operators 2-17
Assignment Operators 2-18
Dynamic Array Operations 2-19

Chapter 3. Compiling BASIC Programs

The BASIC Command 3-1
Compiling Programs in the Background 3-1
BASIC Options 3-2
Compiler Directives 3-4
Including Other Programs 3-4
Replacing and Removing Identifiers 3-5
Selecting Flavor Compatibility 3-6
Specifying Conditional Compilation 3-6
Warnings and Error Messages 3-10
Successful Compilation 3-10
The RUN Command 3-11
Cataloging a BASIC Program 3-11
Catalog Space 3-12
The CATALOG Command 3-13
Deleting Cataloged Programs 3-13
Catalog Shared Memory 3-14

Chapter 4. Locks, Transactions, and Isolation Levels

Locks 4-1
- Shared Record Lock 4-2
- Update Record Lock 4-3
- Shared File Lock 4-4
- Intent File Lock 4-4
- Exclusive File Lock 4-5
- Deadlocks 4-5

Transactions 4-6
- Transaction Terminology 4-6
- Transactions and Data Visibility 4-7
- Transaction Properties 4-7
- Transactions and Locks 4-8
- Transactions and Isolation Levels 4-9
- Using Transactions in BASIC 4-9
- @Variables 4-10
- Transaction Restrictions 4-10

Isolation Levels 4-11
- Data Anomalies 4-12
- Using the ISOMODE Configurable Parameter 4-13
- Isolation Levels and Locks 4-14
- Example 4-15

Chapter 5. Debugging Tools

RAID 5-1
- Invoking RAID from the Command Processor 5-2
- Invoking RAID from a BASIC Program 5-2
- Invoking RAID Using the Break Key 5-3
- Referencing Variables Through RAID 5-3
- RAID Commands 5-4

VLIST 5-12

Appendix A. Quick Reference

Compiler Directives A-1

Declarations A-3

Assignments A-3
Program Flow Control A-3
File I/O A-5
Sequential File I/O A-7
Printer and Terminal I/O A-8
Tape I/O A-10
Select Lists A-10
String Handling A-11
Data Conversion and Formatting A-14
Mathematical Functions A-15
Relational Functions A-18
System A-19
Remote Procedure Calls A-19
Miscellaneous A-20

Appendix B. ASCII and Hex Equivalents

Appendix C. Correlatives and Conversions

Appendix D. BASIC Reserved Words

Appendix E. @Variables

Appendix F. BASIC Subroutines

Index

Preface

This manual describes the features of the UniVerse BASIC programming language. It is intended for experienced programmers. If you have never used BASIC, read Chapter 1 and Chapter 2 before you begin. Before using any statement or function, thoroughly read its description in *UniVerse BASIC Reference*.

If you have previously used a BASIC programming language, you can skim through the first two chapters to determine the difference that may exist between UniVerse BASIC and the BASIC you have used in the past.

The book formerly known as *UniVerse BASIC* is now in three volumes: this volume, *UniVerse BASIC Programmer's Guide*, and the two reference volumes in *UniVerse BASIC Reference*.

Organization of This Manual

This manual contains the following:

Chapter 1 covers information you should know before you begin to use BASIC, such as initial procedures, terminology, and features that are unique to this implementation of BASIC.

Chapter 2 describes types of data, such as constants and variables, and types of operators.

Chapter 3 describes the UniVerse BASIC compiler. The discussion includes instructions on how to run the compiler, compiling options, warnings and error messages, and other related commands.

Chapter 4 describes how to use locks, transaction processing, and isolation levels to prevent data loss and other data conflicts.

Chapter 5 describes the tools available for debugging UniVerse BASIC programs. Included is an interactive debugger, RAID, and the program listing command, VLIST.

The appendixes include tables and charts of the following:

- Quick reference for BASIC statements and functions grouped according to use
- ASCII character codes and equivalents and hexadecimal equivalents
- Correlative and conversion codes
- BASIC reserved words
- Quick reference for @variables
- BASIC subroutines

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
Bold	In syntax, bold indicates commands, function names, keywords, and options that must be input exactly as shown. In text, bold indicates keys to press, function names, and menu selections.
UPPERCASE	In syntax, uppercase indicates UniVerse commands, keywords, and options; BASIC statements and functions; SQL statements and keywords; and ProVerb commands. In text, uppercase also indicates UniVerse identifiers such as filenames, account names, schema names, and record IDs.
Italic	In syntax, italic indicates information that you supply. In text, italic also indicates operating system commands and options, filenames, and pathnames.
`Courier`	Courier indicates examples of source code and system output.
`Courier Bold`	In examples, courier bold indicates characters that the user types or keys the user presses (for example, **`<Return>`**).
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA | itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.

Convention	Usage
➤	A right arrow between menu options indicates you should choose each option in sequence. For example, “Choose **File ➤ Exit**” means you should choose **File** from the menu bar, then choose **Exit** from the File pull-down menu.
`I`	Item mark. For example, the item mark (`I`) in the following string delimits elements 1 and 2, and elements 3 and 4: `1I2F3I4v5`
`F`	Field mark. For example, the field mark (`F`) in the following string delimits elements FLD1 and VAL1: `FLD1FVAL1vSUBV1sSUBV2`
`v`	Value mark. For example, the value mark (`v`) in the following string delimits elements VAL1 and SUBV1: `FLD1FVAL1vSUBV1sSUBV2`
`s`	Subvalue mark. For example, the subvalue mark (`s`) in the following string delimits elements SUBV1 and SUBV2: `FLD1FVAL1vSUBV1sSUBV2`
`T`	Text mark. For example, the text mark (`T`) in the following string delimits elements 4 and 5: `1F2s3v4T5`

The following conventions are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

UniVerse Documentation

UniVerse documentation includes the following:

UniVerse BASIC Programmer’s Guide: This guide contains fundamental information about the UniVerse BASIC language, including data types, variables, operators, debugger commands, and conversion codes. This book is for experienced programmers.

UniVerse BASIC Reference: This two-volume manual contains reference pages for all UniVerse BASIC statements and functions, with complete syntax details. This book is for experienced programmers.

UniVerse BASIC SQL Client Interface Guide: This guide describes how to use the BASIC SQL Client Interface, which provides a programming and a user interface for UniVerse clients to access local or remote UniVerse database servers, and non-UniVerse database servers such as ORACLE, SYBASE, and INFORMIX. This book is for experienced SQL programmers.

Administering UniVerse from UNIX: This manual covers tasks performed by UniVerse Administrators using UNIX shell commands, the UniVerse System Administration menus on UNIX systems, or both. It describes such tasks as system startup and shutdown, system configuration and maintenance, system security, and peripheral maintenance.

Administering UniVerse from Windows: This manual covers tasks performed by UniVerse Administrators using the UniVerse Admin client program on a Windows system. It describes such tasks as maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, and managing file and record locks, network services, and transaction logging.

UniVerse Transaction Logging and Recovery: This manual describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

UniVerse System Description: This manual provides detailed and advanced information about UniVerse features and capabilities for experienced users. It describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

UniVerse User Reference: This manual contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

Guide to RetrieVe: This guide describes RetrieVe, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

Guide to ProVerb: This guide is for application developers who want to use the ProVerb processor to create prestored procedures called procs. It describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

Guide to the UniVerse Editor: This guide describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. It also includes reference pages for all UniVerse Editor commands.

UniVerse SQL Administration for DBAs: This manual is for database administrators (DBAs) who are familiar with UniVerse. It describes tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases.

UniVerse SQL User Guide: This guide is for application developers who are familiar with UniVerse and want to use SQL functionality in their UniVerse applications.

UniVerse SQL Reference: This manual contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

UniVerse Master Index: This book is a comprehensive index for UniVerse documentation.

UniVerse Quick Reference: This manual is a quick reference to all UniVerse commands and keywords. It also summarizes UniVerse SQL statements and keywords, all elements of the UniVerse BASIC language, Editor commands, ProVerb commands, file types, file dictionaries, and user exits.

Related Documentation

The following documentation is also available:

UniVerse Guide for Pick Users: This guide describes UniVerse for new UniVerse users familiar with Pick-based systems.

Moving to UniVerse from PI/open: This manual describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. It includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

UniVerse GCI Guide: This guide describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from BASIC programs. It is for experienced programmers who are familiar with UniVerse.

UniVerse ICI Guide: This guide describes how to use the Integrated Calling Interface (ICI) to access UniVerse from external programs. It is for experienced programmers who are familiar with UniVerse.

UV/Net II Guide: This guide describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. It is for experienced UniVerse administrators.

UV/Term Guide: This guide gives step-by-step instructions for how to install the UV/Term software, how to define terminal parameters, and how to start up and use UV/Term. It also describes the emulator window and how to use it to configure a graphical user interface.

Accessing UniVerse from UNIFACE: This guide describes how to access a UniVerse database from a UNIFACE application. It is for experienced UNIFACE application developers who are new to UniVerse.

UniVerse Call Interface Guide: This guide describes how to use the UniVerse Call Interface (UCI) to write C-based client programs that use SQL function calls to access UniVerse databases. This book is for experienced SQL programmers.

1

Introduction to UniVerse BASIC

UniVerse BASIC is a business-oriented programming language designed to work efficiently with the UniVerse environment. It is easy for a beginning programmer to use yet powerful enough to meet the needs of an experienced programmer.

The power of UniVerse BASIC comes from statements and built-in functions that take advantage of the extensive database management capabilities of UniVerse. These benefits combined with other BASIC extensions result in a development tool well-suited for a wide range of applications.

The extensions in UniVerse BASIC include the following:

- Optional statement labels (that is, statement numbers)
- Statement labels of any length
- Multiple statements allowed on one line
- Computed GOTO statements
- Complex IF statements
- Multiline IF statements
- Priority CASE statement selection
- String handling with variable length strings up to 2^{32}–1 characters
- External subroutine calls
- Direct and indirect subroutine calls
- Magnetic tape input and output
- RetrieVe data conversion capabilities
- UniVerse file access and update capabilities
- File-level and record-level locking capabilities
- Pattern matching
- Dynamic arrays

BASIC Terminology

UniVerse BASIC programmers should understand the meanings of the following terms:

- BASIC program
- Source code
- Object code
- Variable
- Function
- Keyword

BASIC Program. A BASIC program is a set of statements directing the computer to perform a series of tasks in a specified order. A BASIC statement is made up of *keywords* and *variables*.

Source Code. Source code is the original form of the program written by the programmer.

Object Code. Object code is compiler output, which can be executed by the UniVerse RUN command or called as a subroutine.

Variable. A variable is a symbolic name assigned to one or more data values stored in memory. A variable's value can be numeric or character string data, the null value, or it can be defined by the programmer, or it can be the result of operations performed by the program. Variable names can be as long as the physical line, but only the first 64 characters are significant. Variable names begin with an alphabetic character and can include alphanumeric characters, periods (.), dollar signs ($), and percent signs (%). Upper- and lowercase letters are interpreted as different; that is, REC and Rec are different variables.

Function. A BASIC intrinsic function performs mathematical or string manipulations on its arguments. It is referenced by its keyword name and is followed by the required arguments enclosed in parentheses. Functions can be used in expressions; in addition, function arguments can be expressions that include functions. UniVerse BASIC contains both numeric and string functions.

- **Numeric functions.** BASIC can perform certain arithmetic or algebraic calculations, such as calculating the sine (SIN), cosine (COS), or tangent (TAN) of an angle passed as an argument.
- **String functions.** A string function operates on ASCII character strings. For example, the TRIM function deletes extra blank spaces and tabs from a character string, and the STR function generates a particular character string a specified number of times.

Keyword. A BASIC keyword is a word that has special significance in a BASIC program statement. The case of a keyword is ignored; for example, READU and readu are the same keyword. For a list of keywords, see Appendix D.

Subroutines

A *subroutine* is a set of instructions that perform a specific task. It is a small program that can be embedded in a program and accessed with a GOSUB statement, or it can be external to the program and accessed with a CALL statement. Common processes are often kept as external subroutines. This lets the programmer access them from many different programs without having to rewrite them.

When a GOSUB or CALL statement is encountered, program control branches to the referenced subroutine. An internal subroutine must begin with a statement label. An external subroutine must begin with a SUBROUTINE statement.

A RETURN statement can be used at the end of a subroutine to return program flow to the statement following the last referenced GOSUB or CALL statement. If there is no corresponding CALL or GOSUB statement, the program halts and returns to the UniVerse command level. If an external subroutine ends before a RETURN statement is encountered, a RETURN is provided automatically.

Note: If an ABORT, STOP, or CHAIN statement is encountered during subroutine execution, program execution aborts, stops, or chains to another BASIC program and control never returns to the calling program.

One or more arguments separated by commas can be passed to the subroutine as an *argument list*. An argument can be a constant, variable, array variable, or expression, each representing an actual value. The SUBROUTINE *argument list* must contain the same number of arguments so that the subroutine can reference the values being passed to it. Arguments are passed to subroutines by passing a pointer to the argument. Therefore, arguments can also be used to return values to the calling program.

Source Syntax

A BASIC source line has the following syntax:

[*label*] *statement* [; *statement* …] **<Return>**

You can put more than one statement on a line. Separate the statements with semicolons.

A BASIC source line can begin with a statement label. It always ends with a carriage return (**Return**). It can contain up to 256 characters and can extend over more than one physical line.

Statement Types

BASIC statements can be used for any of the following purposes:

- Input and output control
- Program control
- Assignment (assigning a value to a variable)
- Specification (specifying the value of a constant)
- Documentation

Input statements indicate where the computer can expect data to come from (for example, the keyboard, a particular file, and so on). *Output statements* control where the data is displayed or stored.

In general, BASIC statements are executed in the order in which they are entered. *Control statements* alter the sequence of execution by branching to a statement other than the next statement, by conditionally executing statements, or by passing control to a subroutine.

Assignment statements assign values to variables, and *specification statements* assign names to constants.

Program documentation is accomplished by including optional *comments* that explain or document various parts of the program. Comments are part of the source code only and are not executable. They do not affect the size of the object code. Comments must begin with one of the following:

REM * ! $*

Any text that appears between a comment symbol and a carriage return is treated as part of the comment. Comments cannot be embedded in a BASIC statement. If you want to put a comment on the same physical line as a statement, you must end the statement with a semicolon (;), then add the comment, as in the following example:

```
IF X THEN
   A = B; REM correctly formatted comment statement
   B = C
END
```

You cannot put comments between multiple statements on one physical line. For example, in the second line of the following program the statement B = C is part of the comment and is not executed:

```
IF X THEN
   A = B; REM The rest of this line is a comment; B = C
END
```

However, you can put comments in the middle of a statement that occupies more than one physical line, as in the following example:

```
A = 1
B = 2
IF A =
   REM comment
   PRINT A
   REM comment
END ELSE PRINT B
```

Statement Labels

A statement label is a unique identifier for a program line. A statement label consists of a string of characters followed by a colon. The colon is optional when the statement label is completely numeric. Like variable names, alphanumeric statement labels begin with an alphabetic character and can include periods (.), dollar signs ($), and percent signs (%). Upper- and lowercase letters are interpreted as different; that is, ABC and Abc are different labels. Statement labels, like variable names, can be as long as the length of the physical line, but only the first 64 characters are significant. A statement label can be put either in front of a BASIC statement or on its own line. The label must be first on the line—that is, the label cannot begin with a space.

Spaces or Tabs

In a program line, spaces or tabs that are not part of a data item are ignored. Therefore you can use spaces or tabs to improve the program's appearance and readability.

Newlines and Sequential File I/O

UniVerse BASIC uses the term *newline* to indicate the character or character sequence that defines where a line ends in a record in a type 1 or type 19 file. The newline differs according to the operating system you are using. On UNIX file

systems, a newline consists of a single LINEFEED character. On Windows NT file systems, a newline consists of the character sequence RETURN + LINEFEED.

UniVerse BASIC handles this difference transparently in nearly every case, but in a few instances the operating system differences become apparent. If you want your program to work on different operating systems, watch sequential file I/O (that is, writing to or reading from type 1 and type 19 files, line by line or in blocks of data). In particular, be aware of the potential differences that occur:

- When moving a pointer through a file
- When reading or writing blocks of data of a specified length

Special Characters

The UniVerse BASIC character set comprises alphabetic, numeric, and special characters. The alphabetic characters are the upper- and lowercase letters of the alphabet. The numeric characters are the digits 0 through 9. The special characters are as follows. Note that most of the special characters are not permitted in a numeric constant or a variable name.

	Space
	Tab
=	Equal sign or assignment symbol
+	Plus sign
–	Minus sign
*	Asterisk, multiplication symbol, or nonexecutable comment
**	Exponentiation
/	Slash or division symbol
^	Up-arrow or exponentiation symbol
(	Left parenthesis
)	Right parenthesis
#	Number (pound or hash) sign or not equal to
$	Dollar sign
!	Exclamation point or nonexecutable comment
[	Left bracket
]	Right bracket
,	Comma (not permitted in numeric data)
.	Period or decimal point

’	Single quotation mark or apostrophe
;	Semicolon
:	Colon or concatenation
&	Ampersand (and)
<	Less than (left angle bracket)
>	Greater than (right angle bracket)
@	At sign
_	Underscore

Storing Programs

BASIC programs are stored as records in type 1 or type 19 files. The program file must exist before you invoke an editor to create a new record to hold your program. Record IDs must follow the conventions for type 1 and type 19 files.

Editing Programs

You can use the UniVerse Editor or any suitable editor, such as *vi* on UNIX or *edit* on Windows NT, to write your programs. You can edit programs from the UniVerse environment or from the operating system.

Editing Programs Within UniVerse

On UNIX, you can invoke *vi* from the UniVerse system prompt using this syntax:

VI *pathname*

pathname is the relative or absolute pathname of the program you want to edit. For example, the program PAYROLL is stored as a record in the file BP. To edit it with *vi*, enter:

```
>VI BP/PAYROLL
```

If you want to use *vi*, or any other editor, directly from UniVerse, you can write a paragraph or VOC entry that invokes your chosen editor. For example, this VOC entry calls *edit* from UniVerse on Windows NT:

```
    EDIT
001 V
002 \win25\edit.com
003 PR
```

Editing Programs Outside UniVerse

When you edit a program outside UniVerse, remember that the UniVerse file holding the programs is implemented as a directory at the operating system level. For example, the YEAR.END program is stored as a record in the BP file in UniVerse. Its operating system pathname is BP\YEAR.END on Windows NT servers and BP/YEAR.END on UNIX servers.

Getting Started

To create and use a BASIC program, follow these steps:

1. Use the CREATE.FILE command to create a type 1 or type 19 UniVerse file to store your BASIC program source. The RUN command uses the filename BP if a filename is not specified, so many people use BP as the name of their general BASIC program file.
2. Use the UniVerse Editor or some other editor to create the source for your BASIC program as a record in the file you created in step 1.
3. Once you have created the record containing your BASIC program source statements, use the BASIC command to compile your program. The BASIC command creates a file to contain the object code output by the compiler. You do not have to know the name of the object file because the program is always referred to by the source filename.
4. If the BASIC compiler detects any errors, use the Editor to correct the source code and recompile using the BASIC command.
5. When your program compiles without any errors, you can execute it using the RUN command. You can use the RAID command to debug your program.

2

Data Types, Variables, and Operators

This chapter gives an overview of the fundamental components of the UniVerse BASIC language. It describes types of data, constants, variables, and how data is combined with arithmetic, string, relational, and logical operators to form expressions.

Types of Data

Although many program languages distinguish different types of data, the UniVerse BASIC compiler does not. All data is stored internally as character strings, and data typing is done contextually at run time. There are three main types of data: character string, numeric, and unknown (that is, the null value).

Character String Data

Character string data is represented internally as a sequence of ASCII characters. Character strings can represent either numeric or nonnumeric data. Their length is limited only by the amount of available memory. Numeric and nonnumeric data can be mixed in the same character string (for example, in an address).

Character String Constants

In BASIC source code, character string constants are a sequence of ASCII characters enclosed in single or double quotation marks, or backslashes (\ ). These marks are not part of the character string value. The length of character string constants is limited to the length of a statement.

Some examples of character string constants are the following:

```
"Emily Daniels"
'$42,368.99'
'Number of Employees'
"34 Cairo Lane"
\"Fred's Place" isn't open\
```

The beginning and terminating marks enclosing character string data must match. In other words, if you begin a string with a single quotation mark, you must end the string with a single quotation mark.

If you use either a double or a single quotation mark within the character string, you must use the opposite kind to begin and end the string. For example, this string should be written:

```
"It's a lovely day."
```

And this string should be written:

```
'Double quotation marks (") enclosing this string would be
   wrong.'
```

The empty string is a special instance of character string data. It is a character string of zero length. Two adjacent double or single quotation marks, or back-slashes, specify an empty string:

'' or "" or \ \

In your source code you can use any ASCII character in character string constants except ASCII character 0 (NUL), which the compiler interprets as an end-of-string character, and ASCII character 10 (linefeed), which separates the logical lines of a program. Use CHAR(0) and CHAR(10) to embed these characters in a string constant.

Numeric Data

All numeric data is represented internally either as floating-point numbers with the full range of values supported by the system's floating-point implementation, or as integers. On most systems the range is from 10^{-307} through 10^{+307} with 15 decimal digits of precision.

Numeric Constants

Numeric constants can be represented in either fixed-point or floating-point form. Commas and spaces are not allowed in numeric constants.

Fixed-point Constants. Fixed-point form consists of a sequence of digits, optionally containing a decimal point and optionally preceded by a plus (+) or minus (–) sign. Some examples of valid fixed-point constants are:

```
12
-132.4
+10428
```

Floating-point Constants. Floating-point form, which is similar to scientific notation, consists of a sequence of digits, optionally preceded by a plus (+) or minus (–) sign representing the mantissa. The sequence of digits is followed by the letter E and digits, optionally preceded by a minus sign, representing the power of 10 exponent. The exponent must be in the range of –307 through +307. Some examples of valid floating-point constants are:

```
1.2E3
-7.3E42
-1732E-4
```

Use the PRECISION statement to set the maximum number of fractional digits that can result from converting numbers to strings.

Unknown Data: The Null Value

The null value has a special run-time data type in UniVerse BASIC. It was added to UniVerse BASIC for compatibility with UniVerse SQL. The SQL null value represents data whose value is unknown.

Note: Do not confuse the null value with the empty string. The empty string is a character string of zero length which is known to have no value. Unlike null, whose value is defined as unknown, the value of the empty string is known. You cannot use the empty string to represent the null value, nor can you use the null value to represent "no value."

Like all other data in UniVerse BASIC, the null value is represented internally as a character string. The string is made up of the single byte CHAR(128). At run time when explicit or implicit dynamic array extractions are executed on this character, it is assigned the data type "null." UniVerse BASIC programs can reference the null value using the system variable @NULL. They can test whether a value is the null value using the ISNULL and ISNULLS functions.

There is no printable representation of the null value. In this manual the symbol λ (lambda) is sometimes used to denote the null value.

Here is an example of the difference between an empty string and the null value. If you concatenate a string value with an empty string, the string value is returned, but if you concatenate a string value with the null value, null is returned.

```
A = @NULL
B = ""
C = "JONES"
X = C:B
Y = C:A
```

The resulting value of X is "JONES", but the value of Y is the null value. When you concatenate known data with unknown data, the result is unknown.

Programmers should also note the difference between the null value—a special constant whose type is “null”—and the stored representation of the null value—the special character CHAR(128) whose type is “string.” BASIC programs can reference the stored representation of null using the system variable @NULL.STR instead of @NULL.

Constants

Constants are data that do not change in value, data type, or length during program execution. Constants can be character strings or numeric strings (in either integer or floating-point form). A character string of no characters—the empty string—can also be a constant.

Variables

Variables are symbolic names that represent stored data values. The value of a variable can be:

- Unassigned
- A string, which can be an alphanumeric character string, a number, or a dynamic array
- A number, which can be fixed-point (an integer) or floating-point
- The null value
- A dimensioned array (that is, a vector or matrix)
- A subroutine name
- A file
- A select list

The value of a variable can be explicitly assigned by the programmer, or it can be the result of operations performed by the program during execution. Variables can change in value during program execution. At the start of program execution, all variables are unassigned. Any attempt to use an unassigned variable produces an error message.

A variable name must begin with an alphabetic character. It can also include one or more digits, letters, periods, dollar signs, or percent signs. Spaces and tabs are not allowed. A variable name can be any length up to the length of the physical line, but only the first 64 characters are significant. A variable name cannot be any of the reserved words listred in Appendix D. In UniVerse, upper- and lowercase characters in a variable name are interpreted differently.

UniVerse BASIC also provides a set of system variables called @variables. Many of these are read-only variables. Read-only @variables cannot be changed by the programmer.

Most variables in BASIC remain available only while the current program or subroutine is running. Unnamed common variables, however, remain available until the program returns to the system prompt. Named common variables and @variables remain available until the user logs out of UniVerse. See the COMMON statement for information about named and unnamed common variables.

Array Variables

An array is a variable that represents more than one data value. There are two types of array: dimensioned and dynamic. Dimensioned arrays can be either standard or fixed. Fixed arrays are provided in PICK, IN2, and REALITY flavor accounts for compatibility with other Pick systems.

Dimensioned Arrays

Each value in a dimensioned array is called an *element* of the array. Dimensioned arrays can be one- or two-dimensional.

A one-dimensional array is called a *vector*. Its elements are arranged sequentially in memory. An element of a vector is specified by the variable name followed by the index of the element enclosed in parentheses. The index of the first element is 1. The index can be a constant or an expression. Two examples of valid vector element specifiers are:

```
A(1)
COST(35)
```

A two-dimensional array is called a *matrix*. The elements of the first row are arranged sequentially in memory, followed by the elements of the second row, and

so on. An element of a matrix is specified by the variable name followed by two indices enclosed in parentheses. The indices represent the row and column position of the element. The indices of the first element are (1,1). Indices can be constants or expressions. The indices used to specify the elements of a matrix that has four columns and three rows are illustrated by the following:

```
1,1     1,2     1,3     1,4

2,1     2,2     2,3     2,4

3,1     3,2     3,3     3,4
```

Two examples of valid matrix element specifiers are:

```
OBJ(3,1)
WIDGET(7,17)
```

Vectors are treated as matrices with a second dimension of 1. COST(35) and COST(35,1) are equivalent specifications and can be used interchangeably.

Both vectors and matrices have a special *zero element* that is used in MATPARSE, MATREAD, and MATWRITE statements. The zero element of a vector is specified by *vector.name*(0), and the zero element of a matrix is specified by *matrix.name*(0,0). Zero elements are used to store fields that do not fit in the dimensioned elements on MATREAD or MATPARSE statements.

Dimensioned arrays are allocated either at compile time or at run time, depending on the flavor of the account. Arrays allocated at run time are called *standard* arrays. Arrays allocated at compile time are called *fixed* arrays. Standard arrays are redimensionable; fixed arrays are not redimensionable and do not have a zero element. All arrays are standard unless the program is compiled in a PICK, IN2, or REALITY flavor account, in which case they are fixed arrays. To use fixed arrays in PIOPEN, INFORMATION and IDEAL flavor accounts, use the STATIC.DIM option of the $OPTIONS statement. To use standard arrays in PICK, IN2, and REALITY flavor accounts, use $OPTIONS –STATIC.DIM.

Dynamic Arrays

Dynamic arrays map the structure of UniVerse file records to character string data. Any character string can be a *dynamic array.* A dynamic array is a character string containing elements that are substrings separated by delimiters. At the highest level these elements are fields separated by field marks (`F`) (ASCII 254). Each field can contain values separated by value marks (`V`) (ASCII 253). Each value can contain subvalues separated by subvalue marks (`S`) (ASCII 252).

A common use of dynamic arrays is to store data that is either read in from or written out to a UniVerse file record. However, UniVerse BASIC includes facilities

for manipulating dynamic array elements that make dynamic arrays a powerful data type for processing hierarchical information independently of UniVerse files.

The number of fields, values, or subvalues in a dynamic array is limited only by the amount of available memory. Fields, values, and subvalues containing the empty string are represented by two consecutive field marks, value marks, or subvalue marks, respectively.

The following character string is a dynamic array with two fields:

```
TOMsDICKsHARRYvBETTYsSUEsMARYFJONESvSMITH
```

The two fields are:

```
TOMsDICKsHARRYvBETTYsSUEsMARY
```

and:

```
JONESvSMITH
```

Conceptually, this dynamic array has an infinite number of fields, all of which are empty except the first two. References made to the third or fourth field, for example, return an empty string.

The first field has two values:

```
TOMsDICKsHARRY
```

and:

```
BETTYsSUEsMARY
```

The first value has three subvalues: TOM, DICK, and HARRY. The second value also has three subvalues: BETTY, SUE, and MARY.

The second field has two values: JONES and SMITH. Each value has one subvalue: JONES and SMITH.

The following character string:

```
NAME AND ADDRESS
```

can be considered a dynamic array containing one field, which has one value, which has one subvalue, all of which are: NAME AND ADDRESS.

The following character string can be considered a dynamic array containing two fields:

```
JONESvSMITHvBROWNF$1.23vv$2.75
```

The first field has three values: JONES, SMITH, and BROWN. The second field has three values: $1.23, an empty string, and $2.75.

Intrinsic functions and operators allow individual subvalues, values, and fields to be accessed, changed, added, and removed.

You can create a dynamic array in two ways: by treating it as a concatenation of its fields, values, and subvalues; or by enclosing the elements of the dynamic array in angle brackets, using the syntax:

array.name < *field#* , *value#* , *subvalue#* >

For example, to create the dynamic array A as:

```
JONESvSMITHF1.23s20v2.50s10
```

you can say:

```
A="JONES":@VM:"SMITH":@FM:1.23:@SM:20:@VM:2.50:@SM:10
```

or you can say:

```
A = ""
A<1,1> = "JONES"
A<1,2> = "SMITH"
A<2,1,1> = 1.23
A<2,1,2> = 20
A<2,2,1> = 2.50
A<2,2,2> = 10
```

The example has two fields. The first field has two values, and the second field has two values. The first value of the second field has two subvalues, and the second value of the second field also has two subvalues.

You must use the following statements to declare that the first field contains the two values JONES and SMITH:

```
A = ""
A<1,1> = "JONES"
A<1,2> = "SMITH"
```

The statement:

```
A = ""
A<1> = "JONES"
```

declares that the first field contains only JONES with no other values or subvalues. Similarly, the statement:

```
A<2,1> = 1.23
```

declares that the first value of the second field is 1.23 with no subvalues. The statements:

```
A<2,2,1> = 2.50
A<2,2,2> = 10
```

declare that the second value of the second field has two subvalues, 2.50 and 10, respectively.

File Variables

A file variable is created by a form of the OPEN statement. Once opened, a file variable is used in I/O statements to access the file. There are two types of file variable: hashed file variable and sequential file variable. File variables can be scalars or elements of a dimensioned array.

Select List Variables

Select list variables are created by a form of the SELECT statement. A select list variable contains a select list and can be used only in READNEXT statements. Unlike other variables, a select list variable cannot be an element of a dimensioned array.

Expressions

An expression is part of a BASIC statement. It can comprise:

- A string or numeric constant
- A variable
- An intrinsic function
- A user-defined function
- A combination of constants, variables, operators, functions, and other expressions

Format Expressions

A format expression formats variables for output. It specifies the size of the field in which data is displayed or printed, the justification (left, right, or text), the number of digits to the right of the decimal point to display, and so on. Format expressions work like the FMT function. The syntax is:

variable format

format is a valid string expression that evaluates to:

[*width*] [*background*] *justification* [*edit*] [*mask*]

Either *width* or *mask* can specify the size of the display field.

background specifies the character used to pad the field (**Space** is the default padding character).

You must specify *justification* as left, right, or text (text left-justifies output, but breaks lines on spaces when possible).

edit specifies how to format numeric data for output, including such things as the number of digits to display to the right of the decimal point, the descaling factor, whether to round or truncate data, and how to indicate positive and negative currency, handle leading zeros, and so on.

mask is a pattern that specifies how to output data.

If a format expression is applied to the null value, the result is the same as formatting an empty string. This is because the null value has no printable representation.

You can use the STATUS function to determine the result of the format operation. The STATUS function returns the following after a format operation:

0 The format operation is successful.

1 The variable is invalid.

2 The format expression is invalid.

Operators

Operators perform mathematical, string, and logical operations on values. Operands are expressions on which operations are performed. BASIC operators are divided into the following categories:

- Arithmetic
- String
- Relational
- Pattern matching
- IF operator
- Logical
- Assignment
- Dynamic array

Arithmetic Operators

Arithmetic operators combine operands comprising one or more variables, constants, or intrinsic functions. Resulting arithmetic expressions can be combined with other expressions almost indefinitely. The syntax of arithmetic expressions is:

expression operator expression

Table 2-1 lists the arithmetic operators used in BASIC, in order of evaluation.

Table 2-1. Arithmetic Operators

Operator	Operation	Sample Expression
–	Negation	–X
^	Exponentiation	X ^ Y
**		X ** Y
*	Multiplication	X * Y
/	Division	X / Y
+	Addition	X + Y
–	Subtraction	X – Y

You can use parentheses to change the order of evaluation. Operations on expressions enclosed in parentheses are performed before those outside parentheses.

The following expression is evaluated as 112 + 6 + 2, or 120:

```
(14 * 8) + 12 / 2 + 2
```

On the other hand, the next expression is evaluated as 14 * 20 / 4, or 280 / 4, or 70:

```
14 * (8 + 12) / (2 + 2)
```

The result of any arithmetic operation involving the null value is the null value. Since the null value is unknown, the result of combining it with anything must also be unknown. So in the following example, B is the null value:

```
A = @NULL
B = 3 + A
```

The values of arithmetic expressions are internally maintained with the full floating-point accuracy of the system.

If a character string variable containing only numeric characters is used in an arithmetic expression, the character string is treated as a numeric variable. That is, the numeric string is converted to its equivalent internal number and then evaluated numerically in the arithmetic expression. For example, the following expression is evaluated as 77:

```
55 + "22"
```

If a character string variable containing nonnumeric characters is used in an arithmetic expression, a warning message appears, and the string is treated as zero. For example, the following expression is evaluated as 85, and a message warns that the data is nonnumeric:

```
 "5XYZ" + 85
```

A BASIC program compiled in an INFORMATION or a PIOPEN flavor account has arithmetic instructions capable of operating on multivalued data. The following statement in a program compiled in an INFORMATION or a PIOPEN flavor account is valid:

```
C = (23:@VM:46) * REUSE(2)
```

In a BASIC program compiled in an IDEAL, PICK, PIOPEN, REALITY, or IN2 flavor account, arithmetic can be performed on string data only if the string can be interpreted as a single-valued number. The previous statement successfully compiles in PICK, PIOPEN, IN2, REALITY, and IDEAL flavor accounts but causes a run-time error. The REUSE function converts 2 to a string which then has to be converted back to a number for the arithmetic operation. This is harmless. The multivalued string cannot be converted to a number and causes a *nonnumeric data* warning.

The IDEAL flavor uses single-valued arithmetic because of the performance penalty incurred by multivalued arithmetic. To perform multivalued arithmetic in IDEAL, PICK, PIOPEN, IN2, and REALITY flavor accounts, use the VEC.MATH option of the $OPTIONS statement.

String Operators

The concatenation operator (: or CAT) links string expressions to form compound string expressions, as follows:

```
'HELLO.  ' : 'MY NAME IS ' : X : ".  WHAT'S YOURS?"
```

or:

```
'HELLO.  ' CAT 'MY NAME IS ' CAT X CAT ".  WHAT'S YOURS?"
```

If, for instance, the current value of X is JANE, these string expressions both have the following value:

```
"HELLO.  MY NAME IS JANE.  WHAT'S YOURS?"
```

Multiple concatenation operations are performed from left to right. Parenthetical expressions are evaluated before operations outside the parentheses.

With the exception of the null value, all operands in concatenated expressions are considered to be string values, regardless of whether they are string or numeric expressions. However, the precedence of arithmetic operators is higher than the concatenation operator. For example:

```
"THERE ARE " : "2" + "2" : "3" : " WINDOWS."
```

has the value:

```
"THERE ARE 43 WINDOWS."
```

The result of any string operation involving the null value is the null value. Since the null value represents an unknown value, the results of operations on that value are also unknown. But if the null value is referenced as a character string containing only the null value (that is, as the string CHAR(128)), it is treated as character string data. For example, the following expression evaluates to null:

```
"A" : @NULL
```

But this expression evaluates to "A<CHAR128>":

```
"A" : @NULL.STR
```

Substring Operator

A *substring* is a subset of contiguous characters of a character string. For example, JAMES is a substring of the string JAMES JONES. JAMES JON is also a substring of JAMES JONES.

You can specify a substring as a variable name or an array element specifier, followed by two values separated by a comma and enclosed in square brackets. The two values specify the starting character position and the length of the substring. The syntax is:

expression **[** [*start*,] *length* **]**

The bold brackets are part of the syntax and must be typed.

If *start* is 0 or a negative number, the starting position is assumed to be 1. If *start* is omitted, the starting position is calculated according to the following formula:

string.length – *substring.length* + 1

This lets you specify a substring consisting of the last *n* characters of a string without having to calculate the string length. So the following substring specification:

```
"1234567890" [5]
```

returns the substring:

```
67890
```

The following example:

```
A="###DHHH#KK"
PRINT A["#",4,1]
```

displays the following output:

```
DHHH
```

Another syntax for removing substrings from a string, similar to the previous syntax, is:

expression **[** *delimiter*, *occurrence*, *fields* **]**

The bold brackets are part of the syntax and must be typed. Use this syntax to return the substring that is located between the stated number of occurrences of the specified delimiter. *fields* specifies the number of successive fields after the specified occurrence of the delimiter that are to be returned with the substring. The delimiter is part of the returned value when successive fields are returned. This syntax performs the same function as the FIELD function.

All substring syntaxes can be used with the assignment operator (=) to replace the value normally returned by the [ ] operator with the value assigned to the variable. For example:

```
A='12345'
A[3]=1212
PRINT "A=",A
```

returns the following:

```
A= 121212
```

Assigning the three-argument syntax of the [ ] operator provides the same functionality as the FIELDSTORE function.

Relational Operators

Relational operators compare numeric, character string, or logical data. The result of the comparison, either true (1) or false (0), can be used to make a decision

regarding program flow (see the IF statement). Table 2-2 lists the relational operators.

Table 2-2. Relational Operators

Operator	Relation	Example
EQ or =	Equality	X = Y
NE or #	Inequality	X # Y
>< or <>	Inequality	X <> Y
LT or <	Less than	X < Y
GT or >	Greater than	X > Y
LE or <= or =< or #>	Less than or equal to	X <= Y
GE or >= or => or #<	Greater than or equal to	X >= Y

When arithmetic and relational operators are both used in an expression, the arithmetic operations are performed first. For example, the expression:

```
X + Y < (T - 1) / Z
```

is true if the value of X plus Y is less than the value of T minus 1 divided by Z.

String comparisons are made by comparing the ASCII values of single characters from each string. The string with the higher numeric ASCII code equivalent is considered to be greater. If all the ASCII codes are the same, the strings are considered equal.

If the two strings have different lengths, but the shorter string is otherwise identical to the beginning of the longer string, the longer string is considered greater.

Note: An empty string is always compared as a character string. It does not equal numeric zero.

A space is evaluated as less than zero. Leading and trailing spaces are significant. If two strings can be converted to numeric, then the comparison is always made numerically.

Some examples of true comparisons are:

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"CL  " > "CL"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "9/14/93"          (where B$ = "8/14/93")
```

The results of any comparison involving the null value cannot be determined—that is, the result of using a relational operator to compare any value to the null value is unknown. You cannot test for the null value using the = (equal) operator, because the null value is not equal to any value, including itself. The only way to test for the null value is to use the functions ISNULL( ) or ISNULLS( ).

Pattern Matching Operators

The pattern matching operator, MATCH, and its synonym, MATCHES, compare a string expression to a pattern. The syntax for a pattern match expression is:

string MATCH[ES] *pattern*

The pattern is a general description of the format of the string. It can consist of text or the special characters X, A, and N preceded by an integer used as a repeating factor. X stands for any characters, A stands for any alphabetic characters, and N stands for any numeric characters. For example, 3N is the pattern for character strings made up of three numeric characters. If the repeating factor is zero, any number of characters will match the string. For example, 0A is the pattern for any number of alphabetic characters, including none.

An empty string matches the following patterns: "0A", "0X", "0N", "...", "", ", or \\.

IF Operator

The IF operator lets you indicate a value conditional upon the truth of another value. The IF operator has the following syntax:

variable = IF *expression* THEN *expression* ELSE *expression*

variable is assigned the value of the THEN expression if the IF expression is true, otherwise it is assigned the value of the ELSE expression. The IF operator is similar to the IF statement, but it can sometimes be more efficient.

Logical Operators

Numeric data, string data, and the null value can function as logical data. Numeric and string data can have a logical value of true or false. The numeric value 0 (zero), is false; all other numeric values are true. Character string data other than an empty string is true; an empty string is false. The SQL null value is neither true nor false. It has the special logical value of null.

Logical operators perform tests on logical expressions. Logical expressions that evaluate to zero or an empty string are false. Logical expressions that evaluate to null are null. Expressions that evaluate to any other value are true.

The logical operators in UniVerse BASIC are:

- AND (or the equivalent &)
- OR (or the equivalent !)
- NOT

The NOT function inverts a logical value.

The operands of the logical operators are considered to be logical data types. Tables 2-3, 2-4, and 2-5 show logical operation results.

Table 2-3. The AND Operator

AND	**TRUE**	**NULL**	**FALSE**
TRUE	TRUE	NULL	FALSE
NULL	NULL	NULL	FALSE
FALSE	FALSE	FALSE	FALSE

Table 2-4. The OR Operator

OR	**TRUE**	**NULL**	**FALSE**
TRUE	TRUE	TRUE	TRUE
NULL	TRUE	NULL	NULL
FALSE	TRUE	NULL	FALSE

Table 2-5. The NOT Operator

NOT	
TRUE	FALSE
NULL	NULL
FALSE	TRUE

Arithmetic and relational operations take precedence over logical operations. UniVerse logical operations are evaluated from left to right (AND statements do not take precedence over OR statements).

Note: The logical value NULL takes the action of false, because the condition is not known to be true.

Assignment Operators

Assignment operators are used in UniVerse BASIC assignment statements to assign values to variables. Table 2-6 shows the operators and their uses.

Table 2-6. Assignment Operators

Operator	Syntax	Description
=	*variable* = *expression*	Assigns the value of expression to *variable* without any other operation specified.
+=	*variable* += *expression*	Adds the value of *expression* to the previous value of *variable* and assigns the sum to *variable*.
–=	*variable* –= *expression*	Subtracts the value of *expression* from the previous value of *variable* and assigns the difference to *variable*.
:=	*variable* := *expression*	Concatenates the previous value of *variable* and the value of *expression* to form a new value for *variable*.

Table 2-7 shows some examples of assignment statements.

Table 2-7. Examples of Assignment Statements

Example	Interpretation
X = 5	This statement assigns the value 5 to the *variable* X.
X += 5	This statement is equivalent to X=X+5. It adds 5 to the value of the *variable* X, changing the value of X to 10 if it was originally 5.
X –= 3	This statement is equivalent to X=X–3. It subtracts 3 from the value of the *variable* X, changing the value of X to 2 if it was originally 5.
X := Y	This statement is equivalent to X=X:Y. If the value of X is 'CON', and the value of Y is 'CATENATE', then the new value of the *variable* X is 'CONCATENATE'.

Dynamic Array Operations

UniVerse BASIC provides a number of special functions that are designed to perform common operations on dynamic arrays.

Vector Functions

Vector functions process lists of data rather than single values. By using the VEC.MATH (or V) option of the $OPTIONS statement, the arithmetic operators (+, –, *, /) can also operate on dynamic arrays as lists of data.

The operations performed by vector functions or operators are essentially the same as those performed by some standard functions or operators. The difference is that standard functions process all variables as single-valued variables, treating delimiter characters as part of the data. On the other hand, vector functions recognize delimiter characters and process each field, value, and subvalue individually. In fact, vector functions process single-valued variables as if they were dynamic arrays with only the first value defined.

Vector functions have been implemented as subroutines for compatibility with existing UniVerse BASIC programs. Each subroutine is assigned a name made up of the function's name preceded by a hyphen. For example, the name of the subroutine that performs the ADDS function is –ADDS. Because the subroutines are cataloged globally, they can be accessed using the method described in the CALL statement.

The first column of Table 2-8 shows the functions for manipulating dynamic arrays that are available with UniVerse BASIC. The second column shows the corresponding instructions to use for single-valued variables. In this table, *m1* and *m2*

represent dynamic arrays; *s1* and *s2* represent single-valued variables; *p1*, *p2*, and so on, represent single-valued parameters. The value of the function is the resulting dynamic array.

Table 2-8. Vector Functions

Vector Function	Corresponding Instruction for Single-valued Field
ADDS (*m1*, *m2*)	*s1* + *s2*
ANDS (*m1*, *m2*)	*s1* AND *s2*
CATS (*m1*, *m2*)	*s1* : *s2*
CHARS (*m1*)	CHAR (*s1*)
COUNTS (*m1*, *p1*)	COUNT (*s1*, *p1*)
DIVS (*m1*, *m2*)	*s1* / *s2*
EQS (*m1*, *m2*)	*s1* EQ *s2*
ISNULLS (*m1*)	ISNULL (*s1*)
NES (*m1*, *m2*)	*s1* NE *s2*
LES (*m1*, *m2*)	*s1* LE *s2*
LTS (*m1*, *m2*)	*s1* LT *s2*
GES (*m1*, *m2*)	*s1* GE *s2*
GTS (*m1*, *m2*)	*s1* GT *s2*
NOTS (*m1*)	NOT (*s1*)
FIELDS (*m1*, *p1*, *p2*, *p3*)	FIELD (*s1*, *p1*, *p2*, *p3*)
FMTS (*m1*, *p1*)	FMT (*s1*, *p1*)
ICONVS (*m1*, *p1*)	ICONV (*s1*, *p1*)
IFS (*m1*, *m2*, *m3*)	IF *s1* THEN *s2* ELSE *s3*
INDEXS (*m1*, *p1*, *p2*)	INDEX (*s1*, *p1*, *p2*)
LENS (*m1*)	LEN (*s1*)
MODS (*m1*, *m2*)	MOD (*s1*, *s2*)
MULS (*m1*, *m1*)	*s1* * *s2*
NUMS (*m1*)	NUM (*s1*)
OCONVS (*m1*, *p1*)	OCONV (*s1*, *p1*)
ORS (*m1*, *m2*)	*s1* OR *s2*

Table 2-8. Vector Functions (Continued)

Vector Function	Corresponding Instruction for Single-valued Field
SEQS (*m1*)	SEQ (*s1*)
STRS (*m1*, *p1*)	STR (*s1*, *p1*)
SPACES (*m1*)	SPACE (*s1*)
SPLICE (*m1*, *p1*, *m2*)	*s1* : *p1* : *s2*
SUBSTRINGS (*m1*, *p1*, *p2*)	*s1* **[***p1*, *p2***]**
SUBS (*m1*, *m1*)	*s1* – *s2*
TRIMS (*m1*)	TRIM (*s1*)

When a function or operator processes two dynamic arrays, it processes the lists in parallel. In other words, the first value of field A and the first value of field B are processed together, then the second value of field A and the second value of field B are processed together, and so on.

Consider the following example:

```
A = 123v456s7890s2468v10F3691v33s12
B = 13v57s912F1234v8
$OPTIONS VEC.MATH
X = A + B
```

First, the function processing isolates the first field of each dynamic array, which can be written as:

```
A <1> = 123v456s7890s2468v10
B <1> = 13v57s912
```

Then the first values of the first fields are isolated:

```
A <1, 1> = 123
B <1, 1> = 13
```

Then the first subvalues of the first values of the first fields are isolated and added:

```
A <1, 1, 1> = 123
B <1, 1, 1> = 13
```

This produces the first subvalue of the first value of the first field of the result:

```
X <1, 1, 1> = 136
```

Since there are no more subvalues in the first value of either first field, the second values of the first fields are isolated:

```
A <1, 2> = 456s7890s2468
B <1, 2> = 57s912
```

The first subvalues of the second values of the first fields are isolated and added:

```
A <1, 2, 1> = 456
B <1, 2, 1> = 57
```

This produces the first subvalue of the second value of the first field of the result:

```
X <1, 2, 1> = 513
```

Next the subvalues:

```
A <1, 2, 2> = 7890
B <1, 2, 2> = 912
```

are isolated and added to produce:

```
X <1, 2, 2> = 8802
```

Then the subvalues:

```
A <1, 2, 3> = 2468
B <1, 2, 3> = ""
```

are isolated and added to produce:

```
X <1, 2, 3> = 2468
```

Since B <1, 2, 3> does not exist, it is equal to an empty string. In arithmetic expressions an empty string equals zero.

Since there are no more subvalues in either second value of the first fields, these values are isolated:

```
A <1, 3> = 10
B <1, 3> = ""
```

Then the subvalues:

```
A <1, 3, 1> = 10
B <1, 3, 1> = ""
```

are isolated and added to produce:

```
X <1, 3, 1> = 10
```

Since there are no more subvalues or values in either first field, the second fields of each dynamic array are isolated and the process repeats down to the subvalue levels. The second fields can be written as follows:

```
A <2> = 3691v33s12
B <2> = 1234v8
```

Then the first values of the second fields are isolated:

```
A <2, 1> = 3691
B <2, 1> = 1234
```

Then the first subvalues of the first values of the second fields are isolated and added:

```
A <2, 1, 1> = 3691
B <2, 1, 1> = 1234
```

This produces the first subvalue of the first value of the second field of the result:

```
X <2, 1, 1> = 4925
```

Then the second values of the second fields are isolated:

```
A <2, 2> = 33s12
B <2, 2> = 8
```

Then the first subvalues of the second values of the second fields are isolated and added:

```
A <2, 2, 1> = 33
B <2, 2, 1> = 8
```

This produces the first subvalue of the second value of the second field of the result:

```
X <2, 2, 1> = 41
```

Then the second subvalues of the second values of the second fields are isolated and added:

```
A <2, 2, 2> = 12
B <2, 2, 2> = ""
```

This produces the second subvalue of the second value of the second field of the result:

```
X <2, 2, 2> = 12
```

Since there are no more elements in either dynamic array, the result is:

```
X <1, 1, 1> = 136
X <1, 2, 1> = 513
X <1, 2, 2> = 8802
X <1, 2, 3> = 2468
X <1, 3, 1> = 10
X <2, 1, 1> = 4925
X <2, 2, 1> = 41
X <1, 2, 2> = 12
```

These elements are put into the resultant dynamic array, separated by the delimiter mark corresponding to the highest levels that are different (for example, X<1,1,1> and X<1,2,1> have different value levels, so they are separated by a value mark). This yields the following:

```
X = 136v513s8802s2468v10F4925v41s12
```

REUSE Function

If two dynamic arrays are processed by the vector functions described in the preceding section, and they contain unequal numbers of fields, values, or subvalues, then zeros or empty strings are added to the shorter list until the two lists are equal.

When you use the REUSE function, the last value in the shorter list is reused until all the elements in the longer list are exhausted or until the next higher delimiter is encountered.

Dynamic Array Operations and the Null Value

In all dynamic array operations an array reference to a null value treats the null value as an unknown structure of the least bounding delimiter level. For example, the extract operator (< >) extracts the requested data element from a dynamic array. The result of extracting any element from the null value itself is also the null value. If the requested dynamic array element is the stored representation of the null value (CHAR(128)), the null value is returned.

Consider the following three cases:

```
X is λ
Y = ^128
Z = ^128vA
```

X is the null value, Y is a dynamic array containing only the character used to represent the null value (CHAR(128)), and Z is a dynamic array containing two values, CHAR(128) and A, separated by a value mark.

If you extract all or part of the dynamic array from X, you get the null value in all cases:

```
X<1> is λ
X<2> is λ
X<1,2> is λ
```

But if you extract all or part of the dynamic array from Y or Z, you get the null value only when the extract operator specifically references that element of the array:

```
Y<1> is λ
Y<2> = ""
Y<1,2> is λ

Z<1> = ^128vA
Z<2> = ""
Z<1,1> is λ
Z<1,2> = A
```

When the dynamic array extraction finds a string made up of only CHAR(128) between two system delimiters or between a system delimiter and an end-of-string character, CHAR(128) is converted to the null value before any other operation is performed.

See the EXTRACT, INSERT, REPLACE, DELETE, REMOVE, and REUSE functions, and the INS, DEL, and REMOVE statements for details about how BASIC dynamic array operations handle the null value.

3

Compiling BASIC Programs

Before you can run a BASIC program, you must compile it with the UniVerse BASIC compiler. The compiler takes your source code as input and produces executable object code.

Use the UniVerse command CREATE.FILE to create a type 1 or type 19 file in which to store the source code of your BASIC programs. You can create and edit the source code with an operating system editor (such as *vi*), the UniVerse Editor, or a combination of the two.

The BASIC Command

To compile a BASIC program, enter the BASIC command at the system prompt in the following syntax:

BASIC *filename* [*program* |*] ... [*options*]

filename is the name of the type 1 or type 19 file containing the BASIC programs to be compiled. You can compile more than one program at a time if you put all the programs in the same file.

Compiling Programs in the Background

Use the PHANTOM command to compile BASIC programs in the background. The output from PHANTOM processes is stored in the file named &PH&. For example, the command:

```
>PHANTOM BASIC BP *
```

compiles all the programs in BP and puts the output in a record named BASIC_*tttt*_*dddd* in the &PH& file (*tttt* and *dddd* are a time and date stamp).

BASIC Options

You can use the following options with the BASIC command:

–I	Suppresses execution of RAID or VLIST on a compiler or a BASIC program.
–LIST or –L	Generates a listing of the program.
–XREF or –X	Generates a cross-reference table of statement labels and variable names used in the program.
–SPOOL or –S	Generates a listing of the program and spools it directly to the printer rather than to a file.
–T	Suppresses the symbol and line number tables that are usually appended to the end of the object file, for run-time error messages.

A listing produced with either the –LIST or the –XREF option is saved in a file whose name is made up of the source filename and a suffixed .L . The record ID of the program listing in the listing file (*filename*.L) is the same as the record ID in the program file (*filename*).

The –I Option

The –I option inhibits the execution of RAID or VLIST on your BASIC program. This lets you bypass subroutines already debugged and provides security to your subroutines.

The –LIST Option

Listings produced with the –LIST option contain all source lines and all lines inserted with the $INSERT or $INCLUDE statements. The listing is saved in a file whose name is made up of the source filename and a suffixed .L . The record ID of the program listing in the listing file is the same as the record ID in the program file.

The –XREF Option

The –XREF option produces an alphabetical cross-reference listing of every label and variable name used in the program. The listing is saved in a file whose name is made up of the source filename and a suffixed .L . The record ID of the program listing in the listing file is the same as the record ID in the program file.

Consider the following example:

```
>BASIC BP DATE.INT -XREF
Compiling: Source = 'PB/DATE.INT', Object = 'BP.O/DATE.NT'


Compilation Complete.
>ED BP.L DATE.INT
13 lines long.

----: P
0001: BP.L/DATE.INT Source Listing
0002:
0003:
0004: Cross Reference Listing
0005:
0006: Variable....... Type.......... References.......................
 ............
0007:
0008: DATE            Local Scalar   0003=    0004
0009:
0010: *  Definition of symbol
0011: =  Assignment of variable
0012: !  Dimension
0013: @  Argument to CALL
```

The listing shows three columns: “Variable,” “Type,” and “References.” “Variable” is the name of the variable or symbol. “Type” is one of the following symbol types:

Local Scalar	
Local Array	
Common Scalar	
Common Array	
Argument	Variable used in SUBROUTINE statement
Array Arg	Variable used in MAT clause
@variable	One of the system @variables
Label	A program label, as in GOTO FOO
Named Common	Name of a named common segment
Predefined EQU	Predefined equate like @FM, @VM, etc.
Equate	User-defined equate

"References" shows the numbers of all lines in the program that refer to the symbol. Each line number can have a symbol after it to indicate what the line contains:

*	Definition of symbol
=	Assignment of variable
!	Dimension of array
@	Argument to CALL statement

The –SPOOL Option

The –SPOOL option lets you direct output to a printer rather than to a file. Program listings can be spooled for printing with this option.

The –SPOOL option is useful when the listing is very long and you need to look at different parts of the program simultaneously.

The –T Option

The –T option suppresses the table of symbols and the table of line numbers that are appended to the end of object files. These tables are used for handling run-time error messages. Suppressing them results in somewhat smaller object files, but run-time error messages do not know the line number or variable involved in the error.

Compiler Directives

There is a set of BASIC statements that direct the behavior of the compiler. Functions performed by compiler directives include: inserting source code from one program into another program during compilation, setting compile-time compatibility with another UniVerse flavor, and specifying a condition for compiling certain parts of a program. Most of these statements are prefixed by a dollar sign ($).

Including Other Programs

Two statements, $INCLUDE (synonyms are #INCLUDE and INCLUDE) and $CHAIN, instruct the compiler to include the source code of another program in the program currently being compiled. $INCLUDE inserts other code in your program during compilation, returning afterward to compile the next statement in your program. $CHAIN also inserts the code of another program, but after doing so it does not continue reading from the original file. Any program statements following a $CHAIN statement are not compiled.

The syntax for both statements is as follows:

$INCLUDE [*filename*] *program*

$CHAIN [*filename*] *program*

If you do not specify *filename*, the included program must be in the same file as the program you are compiling.

If *program* is in a different file, you must specify *filename* in the $INCLUDE statement. *filename* must be defined in the VOC file.

The $INSERT statement is included for compatibility with Prime INFORMATION programs. $INSERT is used, like $INCLUDE, to insert other code in your program during compilation, returning afterward to compile the next statement in your program.

The syntax for the $INSERT statement is as follows:

$INSERT *primos.pathname*

The PRIMOS pathname is converted to a valid filename using the following conversion rules:

/	converts to	?\
?	converts to	??
An ASCII NUL	converts to	?0
An initial .	converts to	?.

Any leading *> is ignored. If a full pathname is specified, the > between directory names changes to a / to yield the following:

[*pathname* /] *program*

$INSERT uses the transformed argument directly as a filename of the file containing the source to be inserted. It does not use the VOC file.

Replacing and Removing Identifiers

You can define and remove identifiers with the $DEFINE and $UNDEFINE statements. $DEFINE defines an identifier that controls program compilation. $UNDEFINE removes the definition of an identifier.

Replacing Identifiers with $DEFINE

The $DEFINE compiler directive has two functions:

- Defining an identifier that can then be used to control the compilation of a program
- Supplying replacement text for an identifier

You can use $DEFINE to define an identifier that controls conditional compilation. This definition occurs irrespective of whether you supply replacement text. An identifier is defined if it has appeared in a $DEFINE compiler directive. Once the identifier is defined, it can be used to control the conditional compilation of a program.

Removing the Identifier Definition

You use the $UNDEFINE statement to remove the definition of *identifier* from the symbol table, provided that it appeared in a previous $DEFINE statement. If the identifier was not previously defined, $UNDEFINE has no effect.

You can also specify conditional compilation with the $UNDEFINE statement. In this syntax, *identifier* is first defined by the $DEFINE compiler directive. Subsequently the $UNDEFINE compiler directive deletes the definition of *identifier* from the symbol table. The $IFDEF compiler directive determines that *identifier* is not defined, so the statements between the $ELSE and $ENDIF compiler directives are compiled.

Selecting Flavor Compatibility

A $OPTIONS statement is a compiler directive used to select compile-time emulation of any UniVerse flavor. By default, the settings are the same as the flavor of the account. Individual options can also be selected in a program, overriding the usual setting. This does not allow object code that has been compiled in one flavor to execute in another. It allows only the emulation of capabilities of one flavor from within another flavor.

Specifying Conditional Compilation

You can specify the conditions under which all or part of a BASIC program is to be compiled, using:

- A modified version of the IF statement
- $ELSE
- $ENDIF
- $IFDEF
- $IFNDEF

Conditional compilation with the modified IF statement is useful for customizing large programs that are to be used by more than one kind of user. It can also reduce the size of the object code and increase program efficiency.

You can use the compiler directives $IFDEF and $IFNDEF to control whether or not sections of a program are compiled. Both of these compiler directives test a given identifier to see if it is currently defined (that is, has appeared in a $DEFINE compiler directive and has not been undefined). If the identifier that appears in a $IFDEF is defined, all the program source lines appearing between the $IFDEF compiler directive and the closing $ENDIF compiler directive are compiled. If the identifier is not defined, all the lines between the $IFDEF compiler directive and the $ENDIF compiler directive are ignored.

The $IFNDEF compiler directive is the complement to the $IFDEF compiler directive. The lines following the $IFNDEF compiler directive are included in the compilation if the identifier is not defined. If the identifier is defined, all of the lines between the $IFNDEF compiler directive and the $ENDIF compiler directive are ignored. $IFDEF and $IFNDEF compiler directives can be nested up to 10 deep.

IF Statements

The syntax of the conditional compilation statement is the same as that of the IF statement with the exception of the test expression, which must be one of the following: $TRUE, $T, $FALSE, or $F. The syntaxes are as follows:

IF $TRUE THEN *statements* ELSE *statements*

IF $T THEN *statements* ELSE *statements*

IF $FALSE THEN *statements* ELSE *statements*

IF $F THEN *statements* ELSE *statements*

The conditional compilation statement can specify a variable name rather than one of the test specifications listed previously. If it does, an EQUATE statement equating the variable to the test specification must appear at the beginning of the source code. For example:

```
EQUATE USER.A LIT "$T"
IF USER.A THEN statements ELSE statements
```

Consider a program that contains debugging statements in its source code. Using the conditional compilation statement, you could create two versions of the object code from the same source: a test version that contains the debugging statements, and a release version without the debugging statements. The following steps produce the two required versions of the program:

1. Include a conditional debugging statement throughout the source code:

   ```
   IF TEST PRINT X,Y
   ```

2. Put the following statement in the source code before any conditional statements:

   ```
   EQUATE TEST LIT "$TRUE"
   ```

3. Compile the source code to produce object code that contains the debugging statements (the test version).

4. Change the EQUATE statement to:

   ```
   EQUATE TEST LIT "$FALSE"
   ```

5. Compile the source code to produce object code that does not contain the debugging statements (the release version).

The $ELSE Compiler Directive

The $ELSE compiler directive introduces the alternative clause of an $IFDEF or $IFNDEF compiler directive. Its syntax is as follows:

$DEFINE *identifier*
.
.
.
$IFNDEF *identifier*
[*statements*]
$ELSE
[*statements*]
$ENDIF

The $IFNDEF compiler directive determines that *identifier* is defined so that the compiler is forced to compile the statements following the $ELSE compiler directive.

The $ENDIF Compiler Directive

The $ENDIF compiler directive marks the end of a conditional compilation block. Its syntax is as follows:

```
$DEFINE identifier
    .
    .
    .
$UNDEFINE identifier
$IFDEF identifier
    [statements]
$ELSE
    [statements]
$ENDIF
```

identifier is not defined so the statements following the $ELSE compiler directive are compiled. All the statements up to the $ENDIF compiler directive are compiled.

The $IFDEF Compiler Directive

$IFDEF tests for the definition of a compile time symbol. Its syntax is as follows:

```
$IFDEF identifier
    [statements]
[[$ELSE]
    [statements]]
$ENDIF
```

If *identifier* is defined by a $DEFINE compiler directive prior to the $IFDEF compiler directive, the statements following the $IFDEF compiler directive and up to the $ELSE compiler directive are compiled. If *identifier* is not defined prior to the $IFDEF compiler directive, the statements between the $ELSE and the $ENDIF compiler directives are compiled.

The $IFNDEF Compiler Directive

$IFNDEF tests for the definition of a compile time symbol. Its syntax is as follows:

$IFNDEF *identifier*
[*statements*]
[[$ELSE]
[*statements*]]
$ENDIF

If *identifier* is not defined by a prior $DEFINE compiler directive, the statements between the $IFNDEF and $ELSE compiler directives are compiled. Alternatively, if *identifier* is defined by a prior $DEFINE compiler directive, the statements between the $ELSE and $ENDIF compiler directives are compiled.

Warnings and Error Messages

As the compiler attempts to compile a program, various warnings and error messages may appear, disclosing problems that exist in the source code (e.g., statement format errors). When an error occurs, compilation aborts. All errors must be corrected before compilation is successful. Warning messages do not stop compilation.

During compilation, the compiler displays an asterisk on the screen for every 10 lines of source code successfully compiled. You can monitor the progress of the compilation process by counting the number of asterisks on the screen at any time. If an error is encountered, a question mark (?) rather than the asterisk (*) is displayed for those 10 lines.

Successful Compilation

When all errors in the source code are corrected, the compiler successfully completes the compilation by producing an object code record. The object code record is stored in a file whose name is made up of the source filename suffixed with .O (*sourcename*.O). The object code record ID is the same as the source file record ID (program name).

For example, if source code record MAIN is stored in a file called BP, executing the compile statement **`BASIC BP MAIN`** compiles the source code in record MAIN in file BP, producing object code that is stored in record MAIN in file BP.O . The compiler creates the object code file if it does not exist.

The RUN Command

After you have successfully compiled your program, you can run it from the UniVerse system level with the RUN command. Enter the RUN command at the system prompt. The syntax is as follows:

RUN [*filename*] *program* [*options*]

The RUN command appends .O to *filename* and executes the record containing object code in *filename*.O . If *filename* is omitted, the default file, BP, is assumed.

program is the name of the source code of the program. *options* can be one or more of the following:

NO.WARN	Suppresses all warning (nonfatal) messages.
NO.PAGE	Turns off automatic paging. Programs that position the cursor with @ functions do not need to disable pagination.
LPTR	Spools program output to the printer rather than to the terminal screen.
KEEP.COMMON	If the program is executed from within a chain, links the unnamed common.
TRAP	Causes the program to enter the interactive debugger, RAID, whenever a nonfatal error occurs.

For example, the following command executes the record MAIN in the BP.O file:

```
>RUN BP MAIN
```

Run-time error messages are printed on the terminal screen as they are encountered, unless the NO.WARN keyword was specified.

Note: Cataloged programs are considered executable files (that is, the RUN command is not required). To run a cataloged program, enter its catalog name at the system prompt.

Cataloging a BASIC Program

You must catalog a BASIC program in order to:

- Use the program as a subroutine in an I-descriptor
- Execute the program without using the RUN command

Catalog Space

There are three ways to catalog a program: locally, normally (standard), and globally. Each has different implications. There is no one best way of cataloging.

Local Cataloging

Local cataloging creates a VOC entry for the program. This entry is a verb that points to the file and record containing the object code for the cataloged program. A locally cataloged program can be accessed only from the account in which it was cataloged, unless you copy the VOC entry for the catalog name to another account.

Since cataloging a program locally only creates a VOC entry pointing to the object file, you need not recatalog the program every time you recompile it.

Normal Cataloging

Normal cataloging copies the specified object record to the system catalog space and gives it a name of the form:

**account*catalog.name*

Normal cataloging also creates a VOC entry for the program. This entry is a verb that contains the name **account*catalog* in field 2. A normally cataloged program can be accessed only from the account in which it was cataloged, unless you copy the VOC entry for the catalog name to another account or specify the full catalog name, including the account prefix.

Since cataloging a program normally copies the object code to the system catalog space, you must recatalog the program every time you recompile it.

Global Cataloging

Global cataloging copies the specified object record into the system catalog space and gives it a name in one of the following forms:

**catalog.name*
–catalog.name
$catalog.name
!catalog.name

VOC entries are not created for globally cataloged programs. They are available to all accounts on the system as soon as they are cataloged. The UniVerse command processor looks in the system catalog space for verbs or external subroutines that have an initial *. The run machine looks in the system catalog space for verbs or subroutines whose names begin with *, –, $, or ! .

Because cataloging a program globally copies the object code to the system catalog space, you must recatalog the program every time you recompile it.

Note: Because the command processor interprets any line beginning with an asterisk and containing blanks as a comment, you cannot use command parameters when you invoke a globally cataloged program. That is, you can use the following command to run the globally catalog program *GLOBAL, but you cannot include arguments in the command line:

```
>*GLOBAL
```

The CATALOG Command

The CATALOG command is used to catalog a compiled BASIC program, either locally or in the system catalog space. The syntax of the CATALOG command is as follows:

CATALOG [*filename*] [[*catalog.name*] *program.name* | *] [*options*]

If you simply enter CATALOG at the system prompt, you are prompted for the argument values, one at a time:

```
>CATALOG
Catalog name or LOCAL =LOCAL
File name             =BP
Program name          =MONTHLY.SALES
```

If you press **Return** at any of the prompts, CATALOG terminates without cataloging anything. FORCE or NOXREF cannot be specified at a prompt. You can specify the LOCAL keyword in response to the prompt for *catalog.name*.

If you do not specify *catalog.name*, CATALOG uses the program name as the catalog name.

Deleting Cataloged Programs

There are two commands for removing a program from the shared catalog space: DELETE.CATALOG and DECATALOG.

DELETE.CATALOG

The DELETE.CATALOG command removes locally, normally, or globally cataloged programs. It has the following syntax:

DELETE.CATALOG *catalog.name*

catalog.name is used to determine if the program is either globally or normally cataloged. If it is, the program is removed from the system catalog. If the program is not in the system catalog, the VOC file is searched for a local catalog entry. If the program is locally cataloged, only the VOC entry is deleted, not the object code.

If a program is running when you try to delete it, the deletion does not take effect until the program terminates.

DECATALOG

The DECATALOG command removes a locally cataloged program. It deletes the object code and removes the catalog entry from the user's VOC file. It has the following syntax:

DECATALOG [*filename* [[*program*]]

This command deletes the object code of *program* from the ".O" portion of *filename*. Use an asterisk (*) in place of *program* to indicate all records in the file. This command can also be executed after building a select list of programs to be decataloged.

Catalog Shared Memory

UniVerse lets you load BASIC programs from the system catalog into shared memory and run them from there. This reduces the amount of memory needed for multiple users to run the same program at the same time. The program also starts a little faster since it is already in memory and does not have to be read from a disk file.

For example, if 21 users are running the same BASIC program at the same time without catalog shared memory, and the program code requires 50 kilobytes of memory, the total amount of memory used by everyone running that program is 21*50, or 1050, kilobytes. On the other hand, if the program is loaded into catalog shared memory, all 21 users can run one copy of the program, which uses only 50 kilobytes of memory. In this example, catalog shared memory saves 1000 kilobytes, or one megabyte, of memory.

Before users can have access to programs running from catalog shared memory, the system administrator must explicitly choose the programs and load them into memory.

4

Locks, Transactions, and Isolation Levels

This chapter describes the UniVerse BASIC mechanisms that prevent lost updates and other problems caused by data conflicts among concurrent users:

- Locks
- Transactions
- Isolation levels

Locks

UniVerse locks control access to records and files among concurrent users. To provide this control, UniVerse supports the following two levels of lock granularity:

- Fine granularity of record locks
- Coarse granularity of file locks

The level at which you acquire a lock is known as granularity. Record locks affect a smaller component, the record, and provide a fine level of granularity, while file locks affect a larger component, the file, and produce a coarse level of granularity.

Lock compatibility determines what your process can access while other processes have locks on records or files. Record locks allow more compatibility because they coexist with other record locks, thus allowing more transactions to take place concurrently. However these “finer-grained” locks provide a lower isolation level. File locks enforce a high isolation level, more concurrency control, but less compatibility.

Lock compatibility decreases and isolation level increases as strength and granularity increase. This may increase the possibility of deadlocks at high isolation

levels. Within each granularity level, the strength of the lock can vary. UniVerse supports the following locks (in order of increasing strength):

- Shared record
- Update record
- Shared file
- Intent file
- Exclusive file

The locks become less compatible as the granularity, strength, and number of locks increase. Therefore, the number of lock conflicts increase, and fewer users can access the records and files concurrently. To maximize concurrency, you should acquire the minimum lock required to perform a BASIC statement for the shortest period of time. The lock can always be promoted to a lock of greater strength or escalated to a coarser level of granularity if needed.

Shared Record Lock

This lock is also called a READL lock, and is displayed as RL in the LIST.READU output. The shared record lock affects other users as follows:

Allows other users to acquire:	**Prevents other users from acquiring:**	**Is ignored if you already own:**
Shared record	Update record	Shared record
Shared file	Exclusive file	Update record
Intent file		Shared file Intent file Exclusive file

The shared record lock can be promoted or escalated as follows:

Promoted to this lock...	**If...**
Update record	No shared record locks are owned by another user No shared file locks are owned by another user No intent file locks are owned by another user
Escalated to this lock...	**If...**
Shared file	No intent file locks are owned by another user No update record locks are owned by another user

Promoted to this lock...	If...
Intent file	No intent file locks are owned by another user All update record locks are owned by you
Exclusive file	No intent file locks are owned by another user All shared and update record locks are owned by you

In UniVerse BASIC, a shared record lock can be acquired with a MATREADL, READL, READVL, or RECORDLOCKL statement, and released with a CLOSE, RELEASE, or STOP statement.

Update Record Lock

This lock is also called a READU lock, and is displayed as RU in the LIST.READU output. The update record lock affects other users as follows:

Allows other users to acquire:	Prevents other users from acquiring:	Is ignored if you already own:
No locks	Shared record Update record Shared file Intent file Exclusive file	Update record Exclusive file

Note: An update record lock you own is incompatible with a shared file lock you own. Be sure to use a LOCKED clause to avoid deadlocks.

The update record lock can be escalated as follows:

Escalated to this lock...	If...
Intent file	All update record locks are owned by you
Exclusive file	All shared and update record locks are owned by you

In UniVerse BASIC, an update record lock can be acquired or escalated from a shared record lock with a MATREADU, READU, READVU, or RECORDLOCKU statement, and released with a CLOSE, DELETE, MATWRITE, RELEASE, STOP, WRITE, or WRITEV statement.

Shared File Lock

This lock is displayed as FS in the LIST.READU output. The shared file lock affects other users as follows:

Allows other users to acquire:	Prevents other users from acquiring:	Is ignored if you already own:
Shared record	Update record	Shared file
Shared file	Intent file Exclusive file	Intent file Exclusive file

Note: A shared file lock you own is incompatible with an update record lock you own. Be sure to use a LOCKED clause to avoid deadlocks.

The shared file lock can be promoted as follows:

Promoted to this lock...	If...
Intent file	No shared file locks are owned by another user
Exclusive file	No shared file locks or shared record locks are owned by another user

In UniVerse BASIC, a shared file lock can be acquired or promoted with a FILE-LOCK statement, and released with a CLOSE, FILEUNLOCK, RELEASE, or STOP statement.

Intent File Lock

This lock is displayed as IX in the LIST.READU output. The intent file lock affects other users as follows:

Allows other users to acquire:	Prevents other users from acquiring:	Is ignored if you already own:
Shared record	Update record Shared file Intent file Exclusive file	Intent file Exclusive file

The intent file lock can be promoted as follows:

Promoted to this lock...	If...
Exclusive file	No shared record locks are owned by another user

In UniVerse BASIC, an intent file lock can be acquired or promoted from a shared file lock with a FILELOCK statement, and released with a CLOSE, FILEUNLOCK, RELEASE, or STOP statement.

Exclusive File Lock

This lock is displayed as EX in the LIST.READU output. The exclusive file lock affects other users as follows:

Allows other users to acquire:	Prevents other users from acquiring:	Is ignored if you already own:
No locks	Shared record Update record Shared file Intent file Exclusive file	Exclusive file

In UniVerse BASIC, an exclusive file lock can be acquired from a shared file lock with a FILELOCK statement, and released with a CLOSE, FILEUNLOCK, RELEASE, or STOP statement.

Deadlocks

Deadlocks occur when two users who acquire locks incrementally try to acquire a lock that the other user owns and the existing lock is incompatible with the requested lock. The following situations can lead to deadlocks:

- Lock promotion from a shared record or shared file lock to a stronger lock
- Lock escalation to file locks when two users try to escalate at the same time

You can configure UniVerse to automatically identify and resolve deadlocks as they occur through the Deadlock Daemon Administration menu, or you can manually fix a deadlock by selecting and aborting one of the deadlocked user processes. You use the deadlock daemon *uvdlockd* to identify and resolve deadlocks. For more information, see *Administering UniVerse from UNIX*.

Transactions

A *transaction* is a group of logically related operations on the database. In a transaction, either the entire sequence of operations or nothing at all is applied to the database. For example, a banking transaction that involves the transfer of funds from one account to another involves two logically related operations: a withdrawal and a deposit. Both operations, or neither, must be performed if the accounts concerned are to remain reconciled.

This section describes the following transaction topics:

- Terminology
- Data visibility
- Properties
- Locks
- Isolation levels
- Using transactions in BASIC
- @variables
- Restrictions

Transaction Terminology

Release 8.3.3 and later support nested transactions. Within any transaction there can be:

- Read and write operations
- Other transactions, or subtransactions, that can contain other operations or other transactions

When a transaction begins, it is known as active, and the first transaction is the active and top-level transaction. If another transaction begins before the top-level transaction commits or rolls back, the new, or child, transaction also exists with the top-level, or parent, transaction. At this point, the parent transaction is no longer the active transaction. The child transaction becomes active, and remains active until:

- It commits or rolls back, in which case the parent transaction becomes active, and the child no longer exists
- Another child transaction begins and thus becomes the active transaction

Only one transaction can be active at any time, although many transactions can exist at once. There can be only one existing transaction per transaction nesting level. The top-level transaction exists at nesting level 1. When no transactions exist, the nesting level is 0.

Transactions and Data Visibility

Transactions let you safeguard database files on which you are performing operations. This takes place by caching the operations in the active transaction until the top-level transaction commits. At this point the cached operations are applied to the database files, and other users can then see the result of the transaction.

When a read operation for a record occurs, the most recent image of the record is returned to the user. This image is retrieved from the active transaction cache. If it is not found there, the parent transactions are then searched. Finally, if it is not found in the parent transactions, the image is retrieved from the database file.

When a child commits, the operations are adopted by the parent transaction. When it rolls back, the operations are discarded, and do not affect the database or the parent transaction.

Transaction Properties

Each transaction must possess properties commonly referred to as the ACID properties:

- Atomicity
- Consistency
- Isolation
- Durability

In nested transactions, child transactions exhibit atomicity, consistency, and isolation properties. They do not exhibit the durability property since the parent adopts its operation when it commits. The operations only affect the database when the top-level transaction commits. Therefore, even though a child transaction commits, its operations and locks can be lost if the parent transaction rolls back.

Atomicity

Either all the actions of a transaction occur successfully or the transaction is nullified by rolling back all operations. The transaction system ensures that all operations performed by a successfully committed transaction are reflected in the database and the effects of a failed transaction are completely undone.

Consistency

A transaction moves the database from one valid state to another valid state, and if the transaction is prematurely terminated, the database is returned to its previous valid state.

Isolation

The actions carried out by a transaction cannot become visible to another transaction until the transaction commits. Also, a transaction should not be affected by the actions of other concurrent transactions. UniVerse provides different levels of isolation among concurrently executing transactions.

Durability

Once a transaction completes successfully, its effects cannot be altered without running a compensating transaction. The changes made by a successful transaction survive subsequent failures of the system.

Serializability

In addition to the ACID properties, SQL standards stipulate that transactions are serializable. *Serializability* means that the effects of a set of concurrent transactions should produce the same results as though the individual transactions were executed in a serial order, and as if each transaction had exclusive use of the system. In UniVerse, serializability can be achieved by using isolation level 4 for all transactions.

Transactions and Locks

Locks acquired before a transaction exists, or outside the active transaction, are inherited by the active transaction. Locks acquired or promoted within a transaction are not released. Instead, they adhere to the following behavior:

- Locks acquired or promoted within a child transaction are adopted by the parent transaction when the child commits.
- Locks acquired within a child transaction are released when the child transaction rolls back.
- Locks promoted within a child transaction are demoted to the level they were before the start of the child transaction when the child rolls back.
- All locks acquired, promoted, or adopted from child transactions are released when the top-level transaction commits or rolls back.

Transactions and Isolation Levels

If you do not specify an isolation level for the top-level transaction, BASIC uses the UniVerse default isolation level 0. You can change this default by using the BASIC SET TRANSACTION ISOLATION LEVEL statement.

If an isolation level is not specified for a child transaction, the isolation level is inherited from the parent transaction. A specified isolation level should be the same or higher than the parent's isolation level. Isolation levels that are lower than the parent transaction's isolation level are ignored, and the parent's isolation level is used. This occurs because a child transaction's operations must take place under the protection of the higher isolation level so that when it merges with the parent transaction its data is still valid.

Using Transactions in BASIC

The compiler enforces the creation of well-formed transactions. A well-formed transaction occurs when data is first locked and then accessed. A BASIC transaction must include the following three statements:

- BEGIN TRANSACTION
- At least one COMMIT or ROLLBACK statement
- END TRANSACTION

If one of these statements is omitted or out of order, the program does not compile.

The run machine also enforces the use of well-formed transactions. A transaction starts when BEGIN TRANSACTION is executed and ends when COMMIT or ROLLBACK is executed. Program execution then continues at the statement following the next END TRANSACTION statement.

The TXMODE configurable parameter sets the transaction logging mode. When TXMODE is set to 1, the log daemon must be active when the program executes a transaction, otherwise a fatal run-time error occurs. If TXMODE is set to 0, no transactions are logged even if the log daemon is active.

The following example shows transactions within a BASIC program:

```
BEGIN TRANSACTION ISOLATION LEVEL 1
*acquire locks and execute database operations
   BEGIN TRANSACTION ISOLATION LEVEL 2
   *acquire locks and execute database operations
    BEGIN TRANSACTION ISOLATION LEVEL 3
    COMMIT
    END TRANSACTION
    BEGIN TRANSACTION ISOLATION LEVEL 0
    *acquire locks and execute database operations
    ROLLBACK
    COMMIT
   END TRANSACTION
```

@Variables

You can use the following @variables to track transaction activity:

- @ISOLATION
- @TRANSACTION
- @TRANSACTION.ID
- @TRANSACTION.LEVEL

The @ISOLATION variable indicates the current transaction isolation level (0, 1, 2, 3, or 4) for the active transaction or the current default isolation level if no transaction exists.

The @TRANSACTION variable is a numeric value that indicates transaction activity. Any nonzero value indicates that a transaction is active. A 0 indicates that no active transaction exists.

The @TRANSACTION.ID variable indicates the transaction number of the active transaction. An empty string indicates that no transaction exists.

The @TRANSACTION.LEVEL variable indicates the transaction nesting level of the active transaction. A 0 indicates that no transaction exists.

Transaction Restrictions

Other than memory and disk space, there is no restriction on the number of nesting levels within transactions. However, it is important to remember that having many levels in a transaction affects system performance.

If, within a transaction, you try to write to a remote file over UV/Net, the WRITE statement fails, the transaction rolls back, the program terminates, and a run-time error message appears.

You cannot use the following statements while a transaction is active (doing so causes a fatal error):

- CLEARFILE
- SET TRANSACTION ISOLATION LEVEL

You cannot use the EXECUTE or PERFORM statements within a transaction to execute most UniVerse commands and SQL statements. However, you can use

EXECUTE and PERFORM to execute the following UniVerse commands and SQL statements within a transaction:

CHECK.SUM	INSERT	SEARCH	SSELECT
COUNT	LIST	SELECT (RetrieVe)	STAT
DELETE (SQL)	LIST.ITEM	SELECT (SQL)	SUM
DISPLAY	LIST.LABEL	SORT	UPDATE
ESEARCH	RUN	SORT.ITEM	
GET.LIST	SAVE.LIST	SORT.LABEL	

If a BASIC statement has an ON ERROR clause and a fatal error occurs, the ON ERROR clause is ignored.

Isolation Levels

Setting a transaction's isolation level helps avoid various data anomalies. UniVerse BASIC lets you set different levels of isolation depending on which data anomalies you want to avoid. Your transaction runs at the specified isolation level because the transaction subsystem verifies that you have acquired the required locks for that isolation level. If you have not done so, the program fails.

You can specify isolation levels with the following BASIC statements:

- BEGIN TRANSACTION
- SET TRANSACTION ISOLATION LEVEL

You can also establish the isolation level for a session by including a BASIC program that sets the isolation level in the LOGIN paragraph. When you do so, you set the default isolation level for all transactions that takes place during that session, such as UniVerse commands and SQL statements. For example, you could include the statement SET TRANSACTION ISOLATION LEVEL 2 in the program to set the isolation level to 2 each time you logged in to the account. This affects SQL statements and BASIC transactions that take place during this session.

Isolation Level Types

UniVerse BASIC provides the following types of isolation level:

Level	Type
0	NO.ISOLATION
1	READ.UNCOMMITTED
2	READ.COMMITTED
3	REPEATABLE.READ
4	SERIALIZABLE

Each level provides a different degree of protection against the anomalies described in the following section. Only level 4 provides true serializability. However, due to performance issues, for a large system it is recommended that you use level 2 rather than 3 or 4 for most programs. UniVerse provides a default of 0 for backward compatibility.

Data Anomalies

Isolation levels provide protection against the following data anomalies or conflicts that can occur when two processes concurrently access the data:

- Lost updates occur when two processes attempt to update an object at the same time. For example, Process A reads a record. Process B reads the same record, adds 10, and rewrites it. Process A adds 20 to the record that it previously read, and rewrites the record. Thus, Process B's update has been lost.
- Dirty reads occur when one process modifies a record and a second process reads the record before the first commits. If the first process terminates and rolls backs, the second process has read data that should not exist.
- Nonrepeatable reads occur when a process is unable to ensure repeatable reads. For example, this can occur if a transaction reads a record, another transaction updates it, then the first transaction rereads it, and gets a different value the second time.
- Phantom writes occur when a transaction selects a set of records based on some selection criteria and another process writes a record that meets those criteria. The first process repeats the same selection and gets a different set of records.

Table 4-1 lists the data anomalies and the isolation levels at which they can occur.

Table 4-1. Levels at Which Anomalies Can Occur

Anomaly	Level 0	Level 1	Level 2	Level 3	Level 4
Lost update	No[1]	No	No	No	No
Dirty read	Yes	Yes	No	No	No
Nonrepeatable read	Yes	Yes	Yes	No	No
Phantom write	Yes	Yes	Yes	Yes	No

1. Lost updates cannot occur if ISOMODE is set to 2 or 1. If ISOMODE is 0, it is possible for a process running at isolation level 0 to cause a lost update in your process.

Using the ISOMODE Configurable Parameter

The ISOMODE parameter controls the minimum locking requirements for each UniVerse system. By enforcing a minimum level of locking, the transaction management subsystem guarantees that no transaction suffers a lost update due to the actions of another transaction. Protection against lost updates is an important property of serializability. You can set ISOMODE to one of the following settings:

Setting	Description
0	Provides backward compatibility. Transactions are not required to use well-formed writes.
1	Enforces well-formed writes in BASIC transactions. This is the default.
2	Enforces well-formed writes in BASIC programs, whether or not they are in a transaction.

The default ISOMODE setting 1 ensures that BASIC transactions obey the locking rules for isolation level 1, as described in Table 4-2. This means a record cannot be written or deleted within a transaction unless the record or file has been locked for update. A write or delete of a locked record is known as a well-formed write.

ISOMODE setting 0 provides compatibility with earlier UniVerse releases that did not enforce the requirement for well-formed writes in transactions. Since transactions should always use well-formed writes, we recommend that you modify any transactions that do not follow this rule as soon as possible, so that you can set ISOMODE to 1.

Setting ISOMODE to 2 enforces all writes and deletes in BASIC to be well-formed. This mode is available so that when converting an application to use transactions, you can determine whether any programs have not yet been converted. You should not use ISOMODE 2 permanently since many UniVerse system programs are not (and need not be) transactional.

Isolation Levels and Locks

Within transactions, you must consider the level of isolation you need to perform the task, since UniVerse uses locks to ensure that the isolation levels are achieved. As the isolation level increases, the granularity of the locks required becomes coarser. Therefore, the compatibility of locks as well as the number of concurrent users accessing the records and files decreases.

The BASIC run machine checks that the user has acquired the necessary locks to perform a BASIC statement. If the minimum locks for the current isolation level are not held by the user, a fatal error results.

The minimum locks required in a transaction to achieve isolation levels that ensure successful file operation are listed in Table 4-2.

Table 4-2. Isolation Level Locking Requirements

Operation	Isolation Level	Minimum Lock
Read	0 NO.ISOLATION	None
	1 READ.UNCOMMITTED	None
	2 READ.COMMITTED	Shared record (RL)
	3 REPEATABLE.READ	Shared record (RL)
	4 SERIALIZABLE	Shared file (FS)
Delete or Write	0 NO.ISOLATION[1]	None or update record (RU)[1]
	1 READ.UNCOMMITTED	Update record (RU)
	2 READ.COMMITTED	Update record (RU)
	3 REPEATABLE.READ	Update record (RU)
	4 SERIALIZABLE	Update record (RU) and intent file (IX)
Select	4 SERIALIZABLE	Intent file (IX)

1. Different ISOODE settings affect the locking rules for isolation level 0.

In the SQL environment, UniVerse automatically acquires the locks it needs to perform SQL DML (data manipulation language) statements. Lock escalation from record locks to file locks takes place if the number of record locks acquired or promoted within a file in a transaction associated with the SQL DML statements exceeds the value of the configurable parameter MAXRLOCK (the default value is 100). This ensures that the record lock tables do not fill to capacity with large multirow statements.

Example

The following example illustrates how isolation levels affect transactions:

> A transaction running at isolation level 2 deletes records for Customer 100 from the file CUST. Within the transaction, it scans the file ORDERS for all orders placed by this customer, and deletes each order.
>
> The portion of the transaction that deletes the orders does not want to lock the ORDERS file unnecessarily.

The following program illustrates how lock escalation takes place:

```
OPEN "CUST" TO CUST ELSE
    STOP "Cannot open CUST file"
END
OPEN "ORDERS" TO ORDERS ELSE
    CLOSE CUST
    STOP "Cannot open ORDERS file"
END
LOCK.COUNT = 0

** escalate record locks into file locks
** when 10 records have been locked LOCK.ESCALATE = 10
BEGIN TRANSACTION ISOLATION LEVEL 2
    READU CUST.REC FROM CUST,100 THEN
        SELECT ORDERS
GET.NEXT.RECORD:
        LOOP
        WHILE READNEXT ORDERS.NO DO
            ** if lock escalation limit has not been met
            ** obtain a shared record lock for the order
            IF LOCK.COUNT < LOCK.ESCALATE THEN
                READL ORDERS.REC FROM ORDERS,ORDERS.NO ELSE
                    GOTO GET.NEXT.RECORD:
                END
                LOCK.COUNT = LOCK.COUNT + 1
            END ELSE
                ** if lock escalation limit has been reached
                ** obtain intent file lock since the file
                ** needs to be updated
```

```
                IF LOCK.COUNT = LOCK.ESCALATE THEN
                    FILELOCK ORDERS,"INTENT"
                END
                READ ORDERS.REC FROM ORDERS,ORDERS.NO ELSE
                    GOTO GET.NEXT.RECORD:
                END
            END
            IF ORDERS.REC<1> = 100 THEN
                IF LOCK.COUNT < LOCK.ESCALATE THEN
                ** promote shared record lock to
                ** an exclusive record lock
                    READU ORDERS.REC FROM ORDERS,ORDERS.NO THEN NULL
                END ELSE
                ** promote intent file lock to
                ** an exclusive file lock
                    IF LOCK.COUNT = LOCK.ESCALATE THEN
                        FILELOCK ORDERS,"EXCLUSIVE"
                    END
                END
                DELETE ORDERS,ORDERS.NO
            END
        REPEAT
        DELETE CUST,100
    END
COMMIT
END TRANSACTION
CLOSE CUST
CLOSE ORDERS
END
```

5
Debugging Tools

UniVerse provides two debugging tools: RAID and VLIST. RAID is an interactive debugger. VLIST is a diagnostic tool that lists source code followed by object code, as well as statistics about your program.

Note: You cannot run RAID or VLIST on programs compiled with the –I option.

RAID

You can use RAID with your UniVerse BASIC programs. RAID is both an object code and a source code debugger—a powerful tool for detecting errors in UniVerse BASIC code. RAID lets you do the following:

- Set and delete breakpoints. You can suspend execution of the program at specified lines by setting breakpoints in the source code. Once RAID suspends program execution, you can examine the source code, change or display variable values, set additional breakpoints, or delete breakpoints.
- Set watchpoints. You can keep track of changes to variable values by setting watchpoints in the source code. When a variable's value changes, RAID can print both the old and new values and the source instruction that caused the change.
- Step through and display source code, line by line or in segments.
- Examine object addresses.
- Display and modify variables.
- Display all the elements of an array in succession.

You can invoke RAID from the command processor, from within a BASIC program, or by pressing the **Break** key while your BASIC program is executing.

Invoking RAID from the Command Processor

To invoke RAID from the command processor, enter the RAID command instead of the RUN command. The syntax for invoking RAID from the command processor is as follows:

RAID [*filename*] *program* [*options*]

filename is the name of the file in which the source code is stored. RAID appends ".O" to *filename* in order to locate and operate on the object code. If you do not specify *filename*, RAID assumes the BP file by default.

program is the name of the record containing the source code of the program.

options can be one or more of the following:

NO.WARN	Suppresses all warning (nonfatal) error messages. If you do not specify NO.WARN, run-time error messages are printed on the terminal screen as they are encountered.
NO.PAGE	Turns off automatic paging. Programs that position the cursor with @ functions need not disable pagination.
LPTR	Spools program output to the printer rather than to the terminal.
KEEP.COMMON	Maintains the value of variables in unnamed common if a CHAIN statement passes control to another BASIC program.
TRAP	Causes RAID to be reentered whenever a nonfatal error occurs.

Use RAID the same way you use RUN. This causes RAID to be invoked just before program execution. For example, the following command executes the file BP.O/MAIN using the RAID debugger:

```
>RAID BP MAIN
```

When you invoke RAID from the command processor, RAID displays the first executable source code instruction, followed by a double colon (::). Enter a RAID command at the :: prompt. To run the program, enter **R** at the :: prompt. To quit RAID, enter **Q**. RAID commands are discussed in detail in "RAID Commands."

Invoking RAID from a BASIC Program

To invoke RAID from a program, include the DEBUG statement in the program. The syntax is as follows:

DEBUG

The DEBUG statement takes no arguments. When the run machine encounters this statement, the program stops execution, displays a double colon (::), and prompts you to enter a RAID command.

You can also enter the debugger while a BASIC program is running by pressing the **Break** key and then selecting the break option D.

Invoking RAID Using the Break Key

To invoke RAID using the **Break** or **Intr** key, press the **Break** key during execution and then select the break option D.

Referencing Variables Through RAID

Enter variable names as they appear in the BASIC source program. They are case-sensitive, so “A” is not the same variable as “a”.

In addition to regular variable names, you can reference some special “register” variables with RAID. UniVerse BASIC object code is executed by the run machine. When you assign a new variable, the run machine allocates memory for that variable and creates a pointer to that memory location. There are special variables that the run machine uses to hold temporary information. These are the “registers” that can be referenced by $R0 through $R*n*, and a matrix address variable referenced by $MATRIX. An arbitrary number of these registers is available, depending on the needs of your program. The appropriate amount is always made available. You never have more than you need.

Note: Unreferenced variables are not carried in the symbol table of BASIC object code. Therefore, RAID can only display the contents of variables referenced in the current subroutine. RAID ignores all unreferenced variables, and treats them as unknown symbols.

Registers hold intermediate values in a program. For example, for the following statement the sum of 3 and 4 is evaluated and placed in $R0:

```
A=B:3+4
```

The object code evaluates the statement as:

```
A=B:$R0
```

The $MATRIX variable is sometimes used as a pointer to a specific element of an array. It avoids the need to locate the element more than once in the same statement. For example, in the REMOVE statement, the following statement allows for

successive system-delimited substrings in the third element of array B to be put in variable A and a delimiter code setting put in variable C:

```
REMOVE A FROM B(3) SETTING C
```

The reference to the third element of array B is put in the $MATRIX pointer. The object code evaluates the statement as follows:

```
REMOVE A FROM $MATRIX SETTING C
```

RAID Commands

You can enter any RAID command from the double colon (::) prompt. RAID commands have the following general syntax:

position command qualifier

position Tells where and how often to execute the RAID command in the program. You can provide one of the following:

- *line* The decimal number of a line of the source code.
- *address* The hexadecimal address of an object code instruction, indicated by a leading 0X.
- *procedure* The name of a procedure in the source code.
- *variable* The name of a variable in the source code. You must specify the variable exactly as it appears in the source code. Variable names are case-sensitive, so “A” is not the same as “a”. Subscript *variable* to indicate an element of an array. For example, A[1,2].
- *n* Indicates the number of times to execute the command.

qualifier Can be either of the following:

- *string* A string of characters to search for or to replace the value of a variable.
- * Indicates a special form of the specified command.

Table 5-1 summarizes the RAID commands.

Table 5-1. RAID Commands

Command	Description
line	Displays the specified line of the source code.
/[*string*]	Searches the source code for *string*.

Table 5-1. RAID Commands (Continued)

Command	Description
B	Sets a RAID breakpoint.
C	Continues program execution.
D	Deletes a RAID breakpoint.
G	Goes to a line or address, and continues program execution.
H	Displays statistics for the program.
I	Displays and executes the next object code instruction.
L	Displays the next line to be executed.
M	Sets watchpoints.
Q	Quits RAID.
R	Runs the program.
S	Steps through the BASIC source code.
T	Displays the call stack trace.
V	Enters verbose mode for the M command.
V*	Prints the compiler version that generated the object code.
W	Displays the current window.
X	Displays the current object code instruction and address.
X*	Displays local run machine registers and variables.
Z	Displays the next 10 lines of source code.
$	Turns on instruction counting.
#	Turns on program timing.
+	Increments the current line.
–	Decrements the current line.
.	Displays object code instruction and address before execution.
variable /	Prints the value of *variable.*
variable!*string*	Changes the value of *variable* to *string.*

line: Displaying Source Code Lines

line displays a line of the source code specified by its line number. Note that this command displays but does not change the current executable line.

/ : Searching for a Substring

Use a slash followed by a string to search the source code for the next occurrence of the substring *string*. The syntax is as follows:

/ [*string*]

The search begins at the line following the current line. If *string* is found, RAID sets the current line to the line containing *string* and displays that line. If you issue the / command without specifying *string*, RAID searches for the last-searched string. This is an empty string if you have not yet specified a string during this RAID session. If RAID encounters the end of file without finding *string*, it starts at the first line and continues the search until it finds *string* or returns to the current line.

B: Setting Breakpoints

Use the B command to set or list RAID breakpoints. There are two syntaxes:

[*address* | *line* | *procedure* [: *line*]] :B

B*

You can set a RAID breakpoint at the current line, an object code address, a BASIC source line number, the beginning of a specified procedure, or a BASIC source line number within a specified procedure. RAID recognizes lines in called subroutines. RAID executes the program up to the breakpoint and then stops and issues the :: prompt. At that point you can issue another RAID command.

The following example sets breakpoints at line 30 and line 60 of the source code, then runs the program. The program stops executing and displays the RAID prompt when it reaches line 30, and again when it reaches line 60.

```
::30B
::60B
::R
```

The B* command lists all currently active breakpoints.

C: Continuing Program Execution

Use the C command to continue program execution until RAID encounters a breakpoint, or until completion. The C command turns off verbose mode (use the V command to enter verbose mode). If the TRAP command line option is used, RAID is entered at every nonfatal error.

D: Deleting Breakpoints

Use the D command to delete RAID breakpoints. There are two syntaxes:

[*address* | *line*] D

D*

You can delete a RAID breakpoint at the current line, an object code address, or a BASIC source line number. If you use the * option, this command deletes all breakpoints. Some BASIC statements produce multiple run machine statements. If you delete a breakpoint by line number, RAID tries to match the starting address of the BASIC source number. A breakpoint set at anything other than the starting address of a BASIC source line must be deleted by its address.

G: Continuing Program Execution from a Specified Place

Use the G command to go to the line or address specified and to execute the program from that point. The syntax is as follows:

address | *line* G

H: Displaying Program Information

Use the H command to display the version of BASIC used to compile the program, the number of constants used, the number of local variables used, the number of subroutine arguments passed, the object code size, and what procedure RAID was in when the program failed (main program versus subroutine).

I: Executing the Next Object Code Instruction

Use the I command to display and execute the next object code instruction. The syntax is as follows:

[*n*] I

If you use the *n* option, RAID displays and executes the next *n* instructions.

L: Displaying the Next Line

Use the L command to display the next line to be executed.

M: Setting Watchpoints

Use the M command to set watchpoints. The syntaxes are as follows:

variable M [; [*variable*] M ...]

variable =VALUE M

variable is a variable found in the symbol table.

VALUE is the value that you want to break.

The second syntax lets you set a watchpoint for a variable set to a specific value.

A watchpoint condition occurs when RAID monitors a variable until the variable's value changes. The program then suspends operation and displays the variable's old value, its new value, and the source instruction that caused the change to occur. If no change occurs, no display appears. This command accepts up to eight variables. To continue monitoring a variable after RAID has already displayed a change to that variable's value, you need only enter M again, not *variable* M.

Q: Quitting RAID

Use the Q command to quit RAID.

R: Running the Program

Use the R command to run the program until RAID encounters a breakpoint, or until completion. The R command is the same as the C command. The R command turns off verbose mode (use the V command to enter verbose mode). If you specify the TRAP command line option, RAID is entered at every nonfatal error.

S: Stepping Through the Source Code

Use the S command to execute the current line and display the next line of source code. Use multiple S commands to step through the program. The syntax is as follows:

[*n*] S [*]

If the line includes a subroutine call, RAID steps into the subroutine. If you use the *n* option, RAID steps through the next *n* lines. If you use the * option, RAID steps around any subroutine call, essentially treating the entire subroutine as a single line. That is, the S* command instructs RAID to display and execute a source line. If the line includes a subroutine call, RAID executes the subroutine and displays the first source line occurring after the subroutine returns.

T: Displaying the Call Stack Trace

Use the T command to display the call stack trace. It displays the names of the routines that have been called up to the current subroutine.

V: Entering Verbose Mode

Use the V command to enter verbose mode for the M command. In verbose mode RAID displays every source code line until a watchpointed variable's value changes. The program then breaks and displays the variable's old value, its new value, and the source code line that caused the change. To use this command, you must follow it with an M command:

```
::V
::variable M
```

The verbose mode remains on until turned off. Use the C, R, or S command to turn off this mode.

V*: Printing the Compiler Version

Use the V* command to print the version of the compiler that generated the BASIC object code.

W: Displaying the Current Window

Use the W command to display the current window. The syntax is as follows:

[*line*] W

A window comprises a group of 10 lines of source code, centered around the current line. For example, if the current line is 4 when you issue the W command, RAID displays the first 10 lines. The W command by itself does not change the current line. If you use the line option, RAID changes the current line to *line* and displays a window centered around that line. For example, if the current line is 14, RAID displays the lines 9–18. Note that this command affects only the current display lines; it does not affect the executable lines. That is, if you are stepping through the code using the S command, RAID considers the current display line to be the last line displayed before issuing the first S command.

X: Displaying the Current Object Code Instruction

Use the X command to display but not execute the current object code instruction and address.

X*: Displaying the Local Run Machine Registers

Use the X command to display the contents of the local run machine registers (if any exist) and list run machine variables. The syntaxes are as follows:

X [*]

X!

The second syntax lets you set a variable to the empty string. The value of X! is displayed with a carriage return immediately following.

RAID displays the contents. The run machine variables are the following:

Inmat	Value set by INMAT function
Col1	Value set by COL1 function
Col2	Value set by COL2 function
Tab	Value set by TABSTOP statement
Precision	Value set by PRECISION statement
Printer	Printer channel, set by PRINTER statement
Psw	Value set of last internal comparison
Lsw	Value set of last lock test
Status	Value set by STATUS function

Z: Displaying Source Code

Use the Z command to display the next 10 lines of source code and establish the last line displayed as the current line. The syntax is as follows:

[*line*] Z

For example, if the current line is 4 when you issue the Z command, RAID displays lines 4–13, and line 13 becomes the current line.

The current window changes each time this command is used, since the last line printed becomes the current line. For example, if the current line is 14 when you issue the Z command, RAID displays lines 14–23. The current line becomes 23.

If you use the line option, the current line becomes *line*, and RAID displays a window with the specified line first. Regardless of the syntax used, the last line printed becomes the current line once you issue this command. Note that this command affects only the current display lines. It does not affect the executable lines.

$: Turning On Instruction Counting

Use the $ command to turn on instruction counting. RAID records the object code instructions used by the program and the number of times each instruction was executed into a record in your &UFD& file called *profile*. The instruction counting stops when RAID encounters the next break point, a DEBUG statement, **Ctrl-C**, or the end of the program.

: Turning On Program Timing

Use the # command to turn on program timing. RAID records the program name, the number of times it is executed, and the total elapsed time spent during execution into a record of your &UFD& file called *timings*. The program timing stops when RAID encounters the next break point, a DEBUG statement, **Ctrl-C**, or the end of the program.

+ : Incrementing the Current Line Number

Use the + command to increment and display the current object code line number. The syntax is as follows:

[*n*] +

If you use the *n* option, RAID adds *n* to the current line. However, the command only displays valid lines. For example, if you start at line 4 and use the command 3+, the command moves to line 7 only if line 7 is a valid line for the code. If line 7 is invalid, RAID moves to the first valid line after line 7. Note that this command does not affect the currently executed line. That is, if you are stepping through the code using the S command, RAID considers the current line to be the line you displayed before issuing the first S command.

– : Decrementing the Current Line Number

Use the – command to decrement and display the current line number. The syntax is as follows:

[*n*] –

If you use the *n* option, RAID subtracts *n* from the current line. However, the command only displays valid lines. For example, if you start at line 14 and use the command 3–, the command moves to line 11 only if line 11 is a valid address for the code. If line 11 is invalid, RAID moves backward to the first valid address before address 11.

. : Displaying the Next Instruction To Be Executed

Use the **.** (period) command to display the next object code instruction and address to be executed.

variable/ : Printing the Value of a Variable

Use a forward slash (/) after a variable name to print the value of the variable. The syntax is as follows:

variable /

When you enter the *variable* / command, RAID displays the variable's value, and indicates whether it is a string, a number, or the null value. To print an array element, you must subscript the variable. If you enter just an array name, RAID displays the length of the X and Y coordinates. Display matrix values by explicitly displaying the first element, then press **Return** to display subsequent elements. For example, after displaying an array element, pressing **Return** displays successive elements in the array (i.e., 1,1 1,2 1,3 2,1 2,2 2,3, etc.). After the last element in the array displays, the indices wrap to display 0,0 and then 1,1.

! : Changing the Value of a Variable

Use the ! (exclamation point) command to change the value of *variable* to *string*. The syntax is as follows:

variable!*string*

To change an array element, you must subscript the variable. This option is not available for debugging programs in shared memory.

VLIST

Use VLIST to display a listing of the object code of a BASIC program. The syntax for VLIST is as follows:

VLIST [*filename*] *program* [**R**]

filename	The name of the file in which the source code of the BASIC program is stored. The default filename is BP.
program	The name of the program you want to list.
R	Displays internal reference numbers for variables and constants rather than source code names and values.

VLIST displays each line of source code followed by the lines of object code it generated. VLIST also displays program statistics.

```
>VLIST BP TO.LIST
Main Program "BP.O/TO.LIST"
Compiler Version: 7.3.1.1
Object Level    : 5
Machine Type    : 1
Local Variables : 1
Subroutine args : 0
Unnamed Common  : 0
Named Common Seg: 0
Object Size     : 34
Source lines    : 4

0001: FOR I = 1 TO 10
0001 0000 : 0F8 move           0  => I
0001 0006 : 098 forincr        I 10 1 0020:

0002: PRINT I
0002 0014 : 130 printcrlf      I

0003: NEXT I
0003 001A : 0C2 jump           0006:

0004: END
0004 0020 : 190 stop
```

A
Quick Reference

This appendix is a quick reference for all UniVerse BASIC statements and functions. The statements and functions are grouped according to their uses:

- Compiler directives
- Declarations
- Assignments
- Program flow control
- File I/O
- Sequential file I/O
- Printer and terminal I/O
- Tape I/O
- Select lists
- String handling
- Data conversion and formatting
- Mathematical functions
- Relational functions
- System
- Remote procedure calls
- Miscellaneous

Compiler Directives

*	Identifies a line as a comment line. Same as the !, $*, and REM statements.
!	Identifies a line as a comment line. Same as the *, $*, and REM statements.
#INCLUDE	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled. Same as the $INCLUDE and INCLUDE statements.

$*	Identifies a line as a comment line. Same as the *, !, and REM statements.
$CHAIN	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled.
$COPYRIGHT	Inserts comments into the object code header. (This statement is supported by UniVerse for compatibility with existing software.)
$DEFINE	Defines a compile time symbol.
$EJECT	Begins a new page in the listing record. (This statement is supported by UniVerse for compatibility with existing software.) Same as the $PAGE statement.
$ELSE	Introduces an alternative clause of $IFDF or $IFNDEF.
$ENDIF	Marks the end of a conditional compilation block.
$IFDEF, $IFNDEF	Tests for the definition of a compile time symbol.
$INCLUDE	Inserts and compiles UniVerse BASIC source code from another program into the program being compiled. Same as the #INCLUDE and INCLUDE statements.
$INSERT	Performs the same operation as $INCLUDE; the only difference is in the syntax. (This statement is supported by UniVerse for compatibility with existing software.)
$LIST	Enables the listing record to be specified at compile time. (This statement is supported by UniVerse for compatibility with existing software.)
$UNDEFINE	Removes the definition for a compile time symbol.
$OPTIONS	Sets compile time emulation of UniVerse flavors.
$PAGE	Begins a new page in the listing record. (This statement is supported by UniVerse for compatibility with existing software.) Same as the $EJECT statement.
$TITLE	Sets the title to be printed at the top of each page in the listing record.
EQUATE	Assigns a symbol as the equivalent of a variable, function, number, character, or string.

INCLUDE	Inserts and includes the specified BASIC source code from another program into the program being compiled. Same as the #INCLUDE and $INCLUDE statements.
NULL	Indicates that no operation is to be performed.
REM	Identifies a line as a comment line. Same as the *, !, and $* statements.

Declarations

COMMON	Defines a storage area in memory for variables commonly used by programs and external subroutines.
DEFFUN	Defines a user-written function.
DIMENSION	Declares the name, dimensionality, and size constraints of an array variable.
FUNCTION	Identifies a user-written function.
PROGRAM	Identifies a program.
SUBROUTINE	Identifies a series of statements as a subroutine.

Assignments

ASSIGNED( )	Determines if a variable is assigned a value.
CLEAR	Assigns a value of 0 to specified variables.
LET	Assigns a value to a variable.
MAT	Assigns a new value to every element of an array with one statement.
UNASSIGNED( )	Determines if a variable is unassigned.

Program Flow Control

ABORT	Terminates all programs and returns to the UniVerse command level.
BEGIN CASE	Indicates the beginning of a set of CASE statements.
CALL	Executes an external subroutine.

CASE	Alters program flow based on the results returned by expressions.
CHAIN	Terminates a BASIC program and executes a UniVerse command.
CONTINUE	Transfers control to the next logical iteration of a loop.
END	Indicates the end of a program or a block of statements.
END CASE	Indicates the end of a set of CASE statements.
ENTER	Executes an external subroutine.
EXECUTE	Executes UniVerse sentences and paragraphs from within the BASIC program.
EXIT	Quits execution of a LOOP...REPEAT loop and branches to the statement following the REPEAT statement.
FOR	Allows a series of instructions to be performed in a loop a given number of times.
GOSUB	Branches to and returns from an internal subroutine.
GOTO	Branches unconditionally to a specified statement within the program or subroutine.
IF	Determines program flow based on the evaluation of an expression.
LOOP	Repeatedly executes a sequence of statements under specified conditions.
NEXT	Defines the end of a FOR...NEXT loop.
ON	Transfers program control to a specified internal subroutine or to a specified statement, under specified conditions.
PERFORM	Executes a specified UniVerse sentence, paragraph, menu, or command from within the BASIC program, and then returns execution to the statement following the PERFORM statement.
REPEAT	Repeatedly executes a sequence of statements under specified conditions.
RETURN	Transfers program control from an internal or external subroutine back to the calling program.
RETURN (*value*)	Returns a value from a user-written function.

STOP	Terminates the current program.
SUBR( )	Returns the value of an external subroutine.
WHILE/UNTIL	Provides conditions under which the LOOP...REPEAT statement or FOR...NEXT statement terminates.

File I/O

AUTHORIZATION	Specifies the effective run-time UID (user identification) number of the program.
BEGIN TRANSACTION	Indicates the beginning of a set of statements that make up a single transaction.
BSCAN	Scans the leaf-nodes of a B-tree file (type 25) or a secondary index.
CLEARFILE	Erases all records from a file.
CLOSE	Writes data written to the file physically on the disk and releases any file or update locks.
COMMIT	Commits all changes made during a transaction, writing them to disk.
DELETE	Deletes a record from a UniVerse file.
DELETEU	Deletes a record from a previously opened file.
END TRANSACTION	Indicates where execution should continue after a transaction terminates.
FILELOCK	Sets a file update lock on an entire file to prevent other users from updating the file until this program releases it.
FILEUNLOCK	Releases file locks set by the FILELOCK statement.
INDICES( )	Returns information about the secondary key indices in a file.
MATREAD	Assigns the data stored in successive fields of a record from a UniVerse file to the consecutive elements of an array.
MATREADL	Sets a shared read lock on a record, then assigns the data stored in successive fields of the record to the consecutive elements of an array.

MATREADU	Sets an exclusive update lock on a record, then assigns the data stored in successive fields of the record to the consecutive elements of an array.
MATWRITE	Assigns the data stored in consecutive elements of an array to the successive fields of a record in a UniVerse file.
MATWRITEU	Assigns the data stored in consecutive elements of an array to the successive fields of a record in a UniVerse file, retaining any update locks set on the record.
OPEN	Opens a UniVerse file to be used in a BASIC program.
OPENPATH	Opens a file to be used in a BASIC program.
PROCREAD	Assigns the contents of the primary input buffer of the proc to a variable.
PROCWRITE	Writes the specified string to the primary input buffer of the proc that called your BASIC program.
READ	Assigns the contents of a record to a dynamic array variable.
READL	Sets a shared read lock on a record, then assigns the contents of the record to a dynamic array variable.
READU	Sets an exclusive update lock on a record, then assigns the contents of the record to a dynamic array variable.
READV	Assigns the contents of a field of a record to a dynamic array variable.
READVL	Sets a shared read lock on a record, then assigns the contents of a field of a record to a dynamic array variable.
READVU	Sets an exclusive update lock on a record, then assigns the contents of a field of the record to a dynamic array variable.
RECORDLOCKED( )	Establishes whether or not a record is locked by a user.
RECORDLOCKL	Sets a shared read-only lock on a record in a file.
RECORDLOCKU	Locks the specified record to prevent other users from accessing it.
RELEASE	Unlocks records locked by READL, READU, READVL, READVU, MATREADL, MATREADU, MATWRITEV, WRITEV, or WRITEVU statements.

ROLLBACK
: Rolls back all changes made during a transaction. No changes are written to disk.

SET TRANSACTION ISOLATION LEVEL
: Sets the default transaction isolation level for your program.

TRANS( )
: Returns the contents of a field in a record of a UniVerse file.

TRANSACTION ABORT
: Discards changes made during a transaction. No changes are written to disk.

TRANSACTION COMMIT
: Commits all changes made during a transaction, writing them to disk.

TRANSACTION START
: Indicates the beginning of a set of statements that make up a single transaction.

WRITE
: Replaces the contents of a record in a UniVerse file.

WRITEU
: Replaces the contents of the record in a UniVerse file without releasing the record lock.

WRITEV
: Replaces the contents of a field of a record in a UniVerse file.

WRITEVU
: Replaces the contents of a field in the record without releasing the record lock.

XLATE( )
: Returns the contents of a field in a record of a UniVerse file.

Sequential File I/O

CLOSESEQ
: Writes an end-of-file mark at the current location in the record and then makes the record available to other users.

CREATE
: Creates a record in a UniVerse type 1 or type 19 file or establishes a path.

FLUSH
: Immediately writes all buffers.

GET
: Reads a block of data from an input stream associated with a device, such as a serial line or terminal.

GETX
: Reads a block of data from an input stream associated with a device, and returns the characters in ASCII hexadecimal format.

NOBUF	Turns off buffering for a sequential file.
OPENSEQ	Prepares a UniVerse file for sequential use by the BASIC program.
READBLK	Reads a block of data from a UniVerse file open for sequential processing and assigns it to a variable.
READSEQ	Reads a line of data from a UniVerse file opened for sequential processing and assigns it to a variable.
SEND	Writes a block of data to a device that has been opened for I/O using OPENDEV or OPENSEQ.
STATUS	Determines the status of a UniVerse file open for sequential processing.
TIMEOUT	Terminates READSEQ or READBLK if no data is read in the specified time.
TTYCTL	Controls sequential file interaction with a terminal device.
TTYGET	Gets a dynamic array of the terminal characteristics of a terminal, line printer channel, or magnetic tape channel.
TTYSET	Sets the terminal characteristics of a terminal, line printer channel, or magnetic tape channel.
WEOFSEQ	Writes an end-of-file mark to a UniVerse file open for sequential processing at the current position.
WRITEBLK	Writes a block of data to a record in a sequential file.
WRITESEQ	Writes new values to the specified record of a UniVerse file sequentially.
WRITESEQF	Writes new values to the specified record of a UniVerse file sequentially and ensures that the data is written to disk.

Printer and Terminal I/O

@( )	Returns an escape sequence used for terminal control.
BREAK	Enables or disables the **Break** key on the keyboard.
CLEARDATA	Clears all data previously stored by the DATA statement.
CRT	Outputs data to the screen.
DATA	Stores values to be used in subsequent requests for data input.

DISPLAY	Outputs data to the screen.
ECHO	Controls the display of input characters on the terminal screen.
FOOTING	Specifies text to be printed at the bottom of each page.
HEADING	Specifies text to be printed at the top of each page.
HUSH	Suppresses all text normally sent to a terminal during processing.
INPUT	Allows data input from the keyboard during program execution.
INPUT @	Positions the cursor at a specified location and defines the length of the input field.
INPUTCLEAR	Clears the type-ahead buffer.
INPUTDISP @	Positions the cursor at a specified location and defines a format for the variable to print.
INPUTERR	Prints a formatted error message from the ERRMSG file on the bottom line of the terminal.
INPUTNULL	Defines a single character to be recognized as the empty string in an INPUT @ statement.
INPUTTRAP	Branches to a program label or subroutine on a TRAP key.
KEYEDIT	Assigns specific editing functions to the keys on the keyboard to be used with the INPUT statement.
KEYEXIT	Specifies exit traps for the keys assigned editing functions by the KEYEDIT statement.
KEYIN( )	Reads a single character from the input buffer and returns it.
KEYTRAP	Specifies traps for the keys assigned specific functions by the KEYEDIT statement.
OPENDEV	Opens a device for input or output.
PAGE	Prints a footing at the bottom of the page, advances to the next page, and prints a heading at the top.
PRINT	Outputs data to the terminal screen or to a printer.

PRINTER CLOSE	Indicates the completion of a print file and readiness for the data stored in the system buffer to be printed on the line printer.
PRINTER ON | OFF	Indicates whether print file 0 is to output to the terminal screen or to the line printer.
PRINTER RESET	Resets the printing options.
PRINTERR	Prints a formatted error message from the ERRMSG file on the bottom line of the terminal.
PROMPT	Defines the prompt character for user input.
TABSTOP	Sets the current tabstop width for PRINT statements.
TERMINFO( )	Accesses the information contained in the *terminfo* files.
TPARM( )	Evaluates a parameterized *terminfo* string.
TPRINT	Sends data with delays to the screen, a line printer, or another specified print file (that is, a logical printer).

Tape I/O

READT	Assigns the contents of the next record from a magnetic tape unit to the named variable.
REWIND	Rewinds the magnetic tape to the beginning of the tape.
WEOF	Writes an end-of-file mark to a magnetic tape.
WRITET	Writes the contents of a record onto magnetic tape.

Select Lists

CLEARSELECT	Sets a select list to empty.
DELETELIST	Deletes a select list saved in the &SAVEDLISTS& file.
GETLIST	Activates a saved select list so it can be used by a READNEXT statement.
READLIST	Assigns an active select list to a variable.
READNEXT	Assigns the next record ID from an active select list to a variable.

SELECT	Creates a list of all record IDs in a UniVerse file for use by a subsequent READNEXT statement.
SELECTE	Assigns the contents of select list 0 to a variable.
SELECTINDEX	Creates select lists from secondary key indices.
SELECTINFO( )	Returns the activity status of a select list.
SSELECT	Creates a sorted list of all record IDs from a UniVerse file.
WRITELIST	Saves a list as a record in the &SAVEDLISTS& file.

String Handling

ALPHA( )	Determines whether the expression is an alphabetic or nonalphabetic string.
CATS( )	Concatenates elements of two dynamic arrays.
CHANGE( )	Substitutes an element of a string with a replacement element.
CHECKSUM( )	Returns a cyclical redundancy code (a checksum value).
COL1( )	Returns the column position immediately preceding the selected substring after a BASIC FIELD function is executed.
COL2( )	Returns the column position immediately following the selected substring after a BASIC FIELD function is executed.
COMPARE( )	Compares two strings for sorting.
CONVERT	Converts specified characters in a string to designated replacement characters.
CONVERT( )	Replaces every occurrence of specified characters in a variable with other specified characters.
COUNT( )	Evaluates the number of times a substring is repeated in a string.
COUNTS( )	Evaluates the number of times a substring is repeated in each element of a dynamic array.
DCOUNT( )	Evaluates the number of delimited fields contained in a string.
DEL	Deletes the specified field, value, or subvalue from a dynamic array.
DELETE( )	Deletes a field, value, or subvalue from a dynamic array.

DOWNCASE( )	Converts all uppercase letters in an expression to lowercase.
DQUOTE( )	Encloses an expression in double quotation marks.
EREPLACE( )	Substitutes an element of a string with a replacement element.
EXCHANGE( )	Replaces one character with another or deletes all occurrences of a specific character.
EXTRACT( )	Extracts the contents of a specified field, value, or subvalue from a dynamic array.
FIELD( )	Examines a string expression for any occurrence of a specified delimiter and returns a substring that is marked by that delimiter.
FIELDS( )	Examines each element of a dynamic array for any occurrence of a specified delimiter and returns substrings that are marked by that delimiter.
FIELDSTORE( )	Replaces, deletes, or inserts substrings in a specified character string.
FIND	Locates a given occurrence of an element within a dynamic array.
FINDSTR	Locates a given occurrence of a substring.
FOLD( )	Divides a string into a number of shorter sections.
GETREM( )	Returns the numeric value for the position of the REMOVE pointer associated with a dynamic array.
GROUP( )	Returns a substring that is located between the stated number of occurrences of a delimiter.
GROUPSTORE	Modifies existing character strings by inserting, deleting, or replacing substrings that are separated by a delimiter character.
INDEX( )	Returns the starting column position of a specified occurrence of a particular substring within a string expression.
INDEXS( )	Returns the starting column position of a specified occurrence of a particular substring within each element of a dynamic array.
INS	Inserts a specified field, value, or subvalue into a dynamic array.

INSERT( )	Inserts a field, value, or subvalue into a dynamic array.
LEFT( )	Specifies a substring consisting of the first *n* characters of a string.
LEN( )	Calculates the length of a string.
LENS( )	Calculates the length of each element of a dynamic array.
LOCATE	Searches a dynamic array for a particular value or string, and returns the index of its position.
LOWER( )	Converts system delimiters that appear in expressions to the next lower-level delimiter.
MATBUILD	Builds a string by concatenating the elements of an array.
MATCHFIELD( )	Returns the contents of a substring that matches a specified pattern or part of a pattern.
MATPARSE	Assigns the elements of an array from the elements of a dynamic array.
QUOTE( )	Encloses an expression in double quotation marks.
RAISE( )	Converts system delimiters that appear in expressions to the next higher-level delimiter.
REMOVE	Removes substrings from a dynamic array.
REMOVE( )	Successively removes elements from a dynamic array. Extracts successive fields, values, etc., for dynamic array processing.
REVREMOVE	Successively removes elements from a dynamic array, starting from the last element and moving right to left. Extracts successive fields, values, etc., for dynamic array processing.
REPLACE( )	Replaces all or part of the contents of a dynamic array.
REUSE( )	Reuses the last value in the shorter of two multivalue lists in a dynamic array operation.
RIGHT( )	Specifies a substring consisting of the last *n* characters of a string.
SETREM	Sets the position of the REMOVE pointer associated with a dynamic array.
SOUNDEX( )	Returns the soundex code for a string.

SPACE( )	Generates a string consisting of a specified number of blank spaces.
SPACES( )	Generates a dynamic array consisting of a specified number of blank spaces for each element.
SPLICE( )	Inserts a string between the concatenated values of corresponding elements of two dynamic arrays.
SQUOTE( )	Encloses an expression in single quotation marks.
STR( )	Generates a particular character string a specified number of times.
STRS( )	Generates a dynamic array whose elements consist of a character string repeated a specified number of times.
SUBSTRINGS( )	Creates a dynamic array consisting of substrings of the elements of another dynamic array.
TRIM( )	Deletes extra blank spaces and tabs from a character string.
TRIMB( )	Deletes all blank spaces and tabs after the last nonblank character in an expression.
TRIMBS( )	Deletes all trailing blank spaces and tabs from each element of a dynamic array.
TRIMF( )	Deletes all blank spaces and tabs up to the first nonblank character in an expression.
TRIMFS( )	Deletes all leading blank spaces and tabs from each element of a dynamic array.
TRIMS( )	Deletes extra blank spaces and tabs from the elements of a dynamic array.
UPCASE( )	Converts all lowercase letters in an expression to uppercase.

Data Conversion and Formatting

ASCII( )	Converts EBCDIC representation of character string data to the equivalent ASCII character code values.
CHAR( )	Converts a numeric value to its ASCII character string equivalent.
CHARS( )	Converts numeric elements of a dynamic array to their ASCII character string equivalents.

DTX( )	Converts a decimal integer into its hexadecimal equivalent.
EBCDIC( )	Converts data from its ASCII representation to the equivalent code value in EBCDIC.
FIX( )	Rounds an expression to a decimal number having the accuracy specified by the PRECISION statement.
FMT( )	Converts data from its internal representation to a specified format for output.
FMTS( )	Converts elements of a dynamic array from their internal representation to a specified format for output.
ICONV( )	Converts data to internal storage format.
ICONVS( )	Converts elements of a dynamic array to internal storage format.
OCONV( )	Converts data from its internal representation to an external output format.
OCONVS( )	Converts elements of a dynamic array from their internal representation to an external output format.
PRECISION	Sets the maximum number of decimal places allowed in the conversion from the internal binary format of a numeric value to the string representation.
SEQ( )	Converts an ASCII character code value to its corresponding numeric value.
SEQS( )	Converts each element of a dynamic array from an ASCII character code to a corresponding numeric value.
XTD( )	Converts a hexadecimal string into its decimal equivalent.

Mathematical Functions

ABS( )	Returns the absolute (positive) numeric value of an expression.
ABSS( )	Creates a dynamic array containing the absolute values of a dynamic array.
ACOS( )	Calculates the trigonometric arc-cosine of an expression.
ADDS( )	Adds elements of two dynamic arrays.

ASIN( )	Calculates the trigonometric arc-sine of an expression.
ATAN( )	Calculates the trigonometric arctangent of an expression.
BITAND( )	Performs a bitwise AND of two integers.
BITNOT( )	Performs a bitwise NOT of two integers.
BITOR( )	Performs a bitwise OR of two integers.
BITRESET( )	Resets one bit of an integer.
BITSET( )	Sets one bit of an integer.
BITTEST( )	Tests one bit of an integer.
BITXOR( )	Performs a bitwise XOR of two integers.
COS( )	Calculates the trigonometric cosine of an angle.
COSH( )	Calculates the hyperbolic cosine of an expression.
DIV( )	Outputs the whole part of the real division of two real numbers.
DIVS( )	Divides elements of two dynamic arrays.
EXP( )	Calculates the result of base 'e' raised to the power designated by the value of the expression.
INT( )	Calculates the integer numeric value of an expression.
FADD( )	Performs floating-point addition on two numeric values. This function is provided for compatibility with existing software.
FDIV( )	Performs floating-point division on two numeric values.
FFIX( )	Converts a floating-point number to a string with a fixed precision. FFIX is provided for compatibility with existing software.
FFLT( )	Rounds a number to a string with a precision of 14.
FMUL( )	Performs floating-point multiplication on two numeric values. This function is provided for compatibility with existing software.
FSUB( )	Performs floating-point subtraction on two numeric values.
LN( )	Calculates the natural logarithm of an expression in base 'e'.
MAXIMUM( )	Returns the element with the highest numeric value in a dynamic array.

MINIMUM( )	Returns the element with the lowest numeric value in a dynamic array.
MOD( )	Calculates the modulo (the remainder) of two expressions.
MODS( )	Calculates the modulo (the remainder) of elements of two dynamic arrays.
MULS( )	Multiplies elements of two dynamic arrays.
NEG( )	Returns the arithmetic additive inverse of the value of the argument.
NEGS( )	Returns the negative numeric values of elements in a dynamic array. If the value of an element is negative, the returned value is positive.
NUM( )	Returns true (1) if the argument is a numeric data type; otherwise, returns false (0).
NUMS( )	Returns true (1) for each element of a dynamic array that is a numeric data type; otherwise, returns false (0).
PWR( )	Calculates the value of an expression when raised to a specified power.
RANDOMIZE	Initializes the RND function to ensure that the same sequence of random numbers is generated after initialization.
REAL( )	Converts a numeric expression into a real number without loss of accuracy.
REM( )	Calculates the value of the remainder after integer division is performed.
RND( )	Generates a random number between zero and a specified number minus one.
SADD( )	Adds two string numbers and returns the result as a string number.
SCMP( )	Compares two string numbers.
SDIV( )	Outputs the quotient of the whole division of two integers.
SIN( )	Calculates the trigonometric sine of an angle.
SINH( )	Calculates the hyperbolic sine of an expression.
SMUL( )	Multiplies two string numbers.

SQRT( )	Calculates the square root of a number.
SSUB( )	Subtracts one string number from another and returns the result as a string number.
SUBS( )	Subtracts elements of two dynamic arrays.
SUM( )	Calculates the sum of numeric data within a dynamic array.
SUMMATION( )	Adds the elements of a dynamic array.
TAN( )	Calculates the trigonometric tangent of an angle.
TANH( )	Calculates the hyperbolic tangent of an expression.

Relational Functions

ANDS( )	Performs a logical AND on elements of two dynamic arrays.
EQS( )	Compares the equality of corresponding elements of two dynamic arrays.
GES( )	Indicates when elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.
GTS( )	Indicates when elements of one dynamic array are greater than corresponding elements of another dynamic array.
IFS( )	Evaluates a dynamic array and creates another dynamic array on the basis of the truth or falsity of its elements.
ISNULL( )	Indicates when a variable is the null value.
ISNULLS( )	Indicates when an element of a dynamic array is the null value.
LES( )	Indicates when elements of one dynamic array are less than or equal to corresponding elements of another dynamic array.
LTS( )	Indicates when elements of one dynamic array are less than corresponding elements of another dynamic array.
NES( )	Indicates when elements of one dynamic array are not equal to corresponding elements of another dynamic array.
NOT( )	Returns the complement of the logical value of an expression.
NOTS( )	Returns the complement of the logical value of each element of a dynamic array.
ORS( )	Performs a logical OR on elements of two dynamic arrays.

System

DATE( )	Returns the internal system date.
DEBUG	Invokes the interactive UniVerse BASIC debugger RAID.
ERRMSG	Prints a formatted error message from the ERRMSG file.
INMAT( )	Used with the MATPARSE, MATREAD, and MATREADU statements to return the number of array elements or with the OPEN statement to return the modulo of a file.
ITYPE( )	Returns the value resulting from the evaluation of an I-descriptor.
LOCK	Sets an execution lock to protect user-defined resources or events from being used by more than one concurrently running program.
NAP	Suspends execution of a BASIC program, pausing for a specified number of milliseconds.
SENTENCE( )	Returns the stored sentence that invoked the current process.
SLEEP	Suspends execution of a BASIC program, pausing for a specified number of seconds.
STATUS( )	Reports the results of a function or statement previously executed.
SYSTEM( )	Checks the status of a system function.
TIME( )	Returns the time in internal format.
TIMEDATE( )	Returns the time and date.
UNLOCK	Releases an execution lock that was set with the LOCK statement.

Remote Procedure Calls

RPC.CALL( )	Sends requests to a remote server.
RPC.CONNECT( )	Establishes a connection with a remote server process.
RPC.DISCONNECT( )	Disconnects from a remote server process.

Miscellaneous

CLEARPROMPTS	Clears the value of the in-line prompt.
EOF(ARG.)	Checks whether the command line argument pointer is past the last command line argument.
FILEINFO	Returns information about the specified file's configuration.
ILPROMPT( )	Evaluates strings containing in-line prompts.
GET(ARG.)	Retrieves a command line argument.
PUT	Outputs a system error message and adds it to the system error message buffer.
SEEK(ARG.)	Moves the command line argument pointer.

B

ASCII and Hex Equivalents

Table B-1 lists binary, octal, hexadecimal, and ASCII equivalents of decimal numbers.

Table B-1. ASCII Equivalents

Decimal	Binary	Octal	Hexadecimal	ASCII
000	00000000	000	00	NUL
001	00000001	001	01	SOH
002	00000010	002	02	STX
003	00000011	003	03	ETX
004	00000100	004	04	EOT
005	00000101	005	05	ENQ
006	00000110	006	06	ACK
007	00000111	007	07	BEL
008	00001000	010	08	BS
009	00001001	011	09	HT
010	00001010	012	0A	LF
011	00001011	013	0B	VT
012	00001100	014	0C	FF
013	00001101	015	0D	CR
014	00001110	016	0E	SO
015	00001111	017	0F	SI
016	00010000	020	10	DLE
017	00010001	021	11	DC1
018	00010010	022	12	DC2
019	00010011	023	13	DC3
020	00010100	024	14	DC4

Table B-1. ASCII Equivalents (Continued)

Decimal	Binary	Octal	Hexadecimal	ASCII
021	00010101	025	15	NAK
022	00010110	026	16	SYN
023	00010111	027	17	ETB
024	00011000	030	18	CAN
025	00011001	031	19	EM
026	00011010	032	1A	SUB
027	00011011	033	1B	ESC
028	00011100	034	1C	FS
029	00011101	035	1D	GS
030	00011110	036	1E	RS
031	00011111	037	1F	US
032	00100000	040	20	SPACE
033	00100001	041	21	!
034	00100010	042	22	"
035	00100011	043	23	#
036	00100100	044	24	$
037	00100101	045	25	%
038	00100110	046	26	&
039	00100111	047	27	'
040	00101000	050	28	(
041	00101001	051	29	)
042	00101010	052	2A	*
043	00101011	053	2B	+
044	00101100	054	2C	,
045	00101101	055	2D	–
046	00101110	056	2E	.
047	00101111	057	2F	/
048	00110000	060	30	0
049	00110001	061	31	1
050	00110010	062	32	2
051	00110011	063	33	3
052	00110100	064	34	4
053	00110101	065	35	5

Table B-1. ASCII Equivalents (Continued)

Decimal	Binary	Octal	Hexadecimal	ASCII
054	00110110	066	36	6
055	00110111	067	37	7
056	00111000	070	38	8
057	00111001	071	39	9
058	00111010	072	3A	:
059	00111011	073	3B	;
060	00111100	074	3C	<
061	00111101	075	3D	=
062	00111110	076	3E	>
063	00111111	077	3F	?
064	01000000	080	40	@
065	01000001	081	41	A
066	01000010	082	42	B
067	01000011	083	43	C
068	01000100	084	44	D
069	01000101	085	45	E
070	01000110	086	46	F
071	01000111	087	47	G
072	01001000	090	48	H
073	01001001	091	49	I
074	01001010	092	4A	J
075	01001011	093	4B	K
076	01001100	094	4C	L
077	01001101	095	4D	M
078	01001110	096	4E	N
079	01001111	097	4F	O
080	01010000	100	50	P
081	01010001	101	51	Q
082	01010010	102	52	R
083	01010011	103	53	S
084	01010100	104	54	T
085	01010101	105	55	U
086	01010110	106	56	V

Table B-1. ASCII Equivalents (Continued)

Decimal	Binary	Octal	Hexadecimal	ASCII
087	01010111	107	57	W
088	01011000	110	58	X
089	01011001	111	59	Y
090	01011010	112	5A	Z
091	01011011	113	5B	[
092	01011100	114	5C	\
093	01011101	135	5D	]
094	01011110	136	5E	^
095	01011111	137	5F	_
096	01100000	140	60	`
097	01100001	141	61	a
098	01100010	142	62	b
099	01100011	143	63	c
100	01100100	144	64	d
101	01100101	145	65	e
102	01100110	146	66	f
103	01100111	147	67	g
104	01101000	150	68	h
105	01110001	151	69	i
106	01110010	152	6A	j
107	01110011	153	6B	k
108	01110100	154	6C	l
109	01110101	155	6D	m
110	01110110	156	6E	n
111	01110111	157	6F	o
112	01111000	160	70	p
113	01111001	161	71	q
114	01111010	162	72	r
115	01111011	163	73	s
116	01111100	164	74	t
117	01110101	165	75	u
118	01110110	166	76	v
119	01110111	167	77	w

Table B-1. ASCII Equivalents (Continued)

Decimal	Binary	Octal	Hexadecimal	ASCII
120	01111000	170	78	x
121	01111001	171	79	y
122	01111010	172	7A	z
123	01111011	173	7B	{
124	01111100	174	7C	|
125	01111101	175	7D	}
126	01111110	176	7E	~
127	01111111	177	7F	DEL
128	10000000	200	80	SQLNULL
251	11111011	373	FB	TM
252	11111100	374	FC	SM
253	11111101	375	FD	VM
254	11111110	376	FE	FM
255	11111111	377	FF	IM

Table B-2 provides additional hexadecimal and decimal equivalents.

Table B-2. Additional Hexadecimal and Decimal Equivalents

Hexadecimal	Decimal	Hexadecimal	Decimal
80	128	3000	12288
90	144	4000	16384
A0	160	5000	20480
B0	176	6000	24576
C0	192	7000	28672
D0	208	8000	32768
E0	224	9000	36864
F0	240	A000	40960
100	256	B000	45056
200	512	C000	49152
300	768	D000	53248
400	1024	E000	57344
500	1280	F000	61440

C

Correlatives and Conversions

This appendix describes the correlative and conversion codes used in dictionary entries and with the ICONV, ICONVS, OCONV, and OCONVS functions in BASIC. Use conversion codes with the ICONV function when converting data to internal storage format and with the OCONV function when converting data from its internal representation to an external output format. Read this entire appendix and both the ICONV function and OCONV function sections before attempting to perform internal or external data conversion.

Note: If you try to convert the null value, null is returned and the STATUS function returns 1 (invalid data).

Table C-1 lists correlative and conversion codes.

Table C-1. Correlative and Conversion Codes

Code	Description
A	Algebraic functions
C	Concatenation
D	Date conversion
ECS	Extended character set conversion
F	Mathematical functions
G	Group extraction
L	Length functions
MB	Binary conversion
MCA	Masked alphabetic conversion

Table C-1. Correlative and Conversion Codes (Continued)

Code	Description
MC/A	Masked nonalphabetic conversion
MCD	Decimal to hexadecimal conversion
MCDX	Decimal to hexadecimal conversion
MCL	Masked lowercase conversion
MCN	Masked numeric conversion
MC/N	Masked nonnumeric conversion
MCP	Masked unprintable character conversion
MCT	Masked initial capitals conversion
MCU	Masked uppercase conversion
MCX	Hexadecimal to decimal conversion
MCXD	Hexadecimal to decimal conversion
MD	Masked decimal conversion
ML	Masked left conversion
MO	Octal conversion
MP	Packed decimal conversion
MR	Masked right conversion
MT	Time conversion
MX	Hexadecimal conversion
MY	ASCII conversion
NR	Roman numeral conversion
P	Pattern matching
Q	Exponential conversion
R	Range function
S	Soundex
S	Substitution
T	Text extraction
T *filename*	File translation

Format

A [;] *expression*

The A code converts A codes into F codes in order to perform mathematical operations on the field values of a record, or to manipulate strings. The A code functions in the same way as the F code but is easier to write and to understand.

expression can be one or more of the following:

A data location or string

loc[R]	Field number specifying a data value, followed by an optional R (repeat code).
N(*name*)	*name* is a dictionary entry for a field. The name is referenced in the file dictionary. An error message is returned if the field name is not found. Any codes specified in field 3 of *name* are applied to the field defined by *name*, and the converted value is processed by the A code.
string	Literal string enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ).
number	Constant number enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ). Any integer, positive, negative, or zero can be specified.
D	System date (in internal format).
T	System time (in internal format).

A special system counter operand

@NI	Current system counter (number of items listed or selected).
@ND	Number of detail lines since the last BREAK on a break line.
@NV	Current value counter for columnar listing only.
@NS	Current subvalue counter for columnar listing only.
@NB	Current BREAK level number. 1 = lowest level break. This has a value of 255 on the grand-total line.
@LPV	**L**oad **P**revious **V**alue: load the result of the last correlative or conversion onto the stack.

A code: Algebraic Functions

A function

R(*exp*)	Remainder after integer division of the first operand by the second. For example, R(2,"5") returns the remainder when field 2's value is divided by 5.
S(*exp*)	Sum all multivalues in *exp*. For example, S(6) sums the multivalues of field 6.
IN(*exp*)	Test for the null value.
[ ]	Extract substring. Field numbers, literal numbers, or expressions can be used as arguments within the brackets. For example, if the value of field 3 is 9, then 7["2",3] returns the second through ninth characters of field 7. The brackets are part of the syntax and must be typed.
IF(*expression*) | THEN(*expression*) | ELSE(*expression*)	
(*conv*)	Conversion expression in parentheses (except A and F conversions).

An arithmetic operator

*	Multiply operands.
/	Divide operands. Division always returns an integer result: for example, "3" / "2" evaluates to 1, not to 1.5.
+	Add operands.
–	Subtract operands.
:	Concatenate operands.

A relational operator

=	Equal to
<	Less than
>	Greater than
# or <>	Not equal to
<=	Less than or equal to
>=	Greater than or equal to

A conditional operator

AND	Logical AND
OR	Logical OR

In most cases F and A codes do not act on a data string passed to them. The code specification itself contains all the necessary data (or at least the names of fields that contain the necessary data). So the following A codes produce identical F codes, which in turn assign identical results to X:

```
X = OCONV( "123", "A;'1' + '2'" )
X = OCONV( "", "A;'1' + '2'" )
X = OCONV( @ID, "A;'1' + '2'" )
X = OCONV( "The quick brown fox jumped over a lazy dog's
   back","A;'1' + '2'" )
```

The data strings passed to the A code—123, the empty string, the record ID, and "The quick brown fox..." string—simply do not come into play. The only possible exception occurs when the user includes the LPV (load previous value) special operand in the A or F code. The following example adds the value 5 and the previous value 123 to return the sum 128:

```
X = OCONV( "123", "A;'5' + LPV" )
```

It is almost never right to call an A or F code using the vector conversion functions OCONVS and ICONVS. In the following example, Y = 123v456v789:

```
X = OCONVS( Y, "A;'5' + '2' )
```

The statement says, "For each value of Y, call the A code to add 5 and 2." (v represents a value mark.) The A code gets called three times, and each time it returns the value 7. X, predictably, gets assigned 7. The scalar OCONV function returns the same result in much less time.

What about correlatives and conversions within an A or F code? Since any string in the A or F code can be multivalued, the F code calls the vector functions OCONVS or ICONVS any time it encounters a secondary correlative or conversion. In the following example, the F code—itself called only once—calls OCONVS to ensure that the G code gets performed on each value of @RECORD< 1 >. X is assigned the result *ccc*v*fff*:

```
@RECORD< 1 > = aaa*bbb*cccvddd*eee*fff
X = OCONV( "", "A;1(G2*1)"
```

The value mark is reserved to separate individual code specifications where multiple successive conversions must be performed.

The following dictionary entry specifies that the substring between the first and second asterisks of the record ID should be extracted, then the first four characters of that substring should be extracted, then the masked decimal conversion should be applied to that substring:

```
001: D
002: 0
003: G1*1vT1,4vMD2
004: Foo
005: 6R
006: S
```

To attempt to define a multivalued string as part of the A or F code itself rather than as part of the @RECORD produces invalid code. For instance, both:

```
X = OCONV( "", "A;'aaa*bbb*cccvddd*eee*fff'(G2*1)" )
```

and the dictionary entry:

```
001: D
002: 0
003: A;'aaa*bbb*cccvddd*eee*fff'(G2*1)
004: Bar
005: 7L
006: S
```

are invalid. The first returns an empty string (the original value) and a status of 2. The second returns the record ID; if the STATUS function were accessible from dictionary entries, it would also be set to 2.

Format

C [;] *expression1 c expression2* [*c expression3* ...]

The C code chains together field values or quoted strings, or both.

The semicolon is optional and is ignored.

c is the character to be inserted between the fields. Any nonnumeric character (except system delimiters) is valid, including a blank. A semicolon (;) is a reserved character that means no separation character is to be used. Two separators cannot follow in succession, with the exceptions of semicolons and blanks.

expression is a field number and requests the contents of that field; or any string enclosed in single quotation marks ('), double quotation marks ("), or back-slashes (\ ); or an asterisk (*), which specifies the data value being converted.

You can include any number of delimiters or expressions in a C code.

Examples

Assume a BASIC program with @RECORD = "oneFtwoFthreeVfour":

Statement	Output
`PRINT OCONV("x","C;1;'xyz';2")`	`onexyztwo`
`PRINT ICONV("x","C;2;'xyz';3")`	`twoxyzthreeVfour`
`PRINT OCONV("","C;2;'xyz';3")`	
`PRINT ICONV(x,"C;1***2")`	`one*x*two`
`PRINT OCONV(0,"C;1:2+3")`	`one:two+threeVfour`

There is one anomaly of the C code (as implemented by ADDS Mentor, at least) that the UniVerse C code does not reproduce:

`PRINT ICONV ( x, "C*1*2*3" )`	`x1x2x3`

The assumption that anything following a nonseparator asterisk is a separator seems egregious, so the UniVerse C code implements:

`PRINT ICONV (x, "C*1*2*3" )`	`xone*two*threeVfour`

Anyone wanting the ADDS effect can quote the numbers.

D code: Date Conversion

Format

D [*n*] [*m] [*s*] [*fmt* **[** *f1* , *f2* , *f3* **]**] [E] [L]

The D code converts input dates from conventional formats to an internal format for storage. It also converts internal dates back to conventional formats for output. When converting an input date to internal format, date conversion specifies the format you use to enter the date. When converting internal dates to external format, date conversion defines the external format for the date.

You can set the default date format with the DATE.FORMAT command. A system-wide default date format can be set in the *msg.text* file of the UV account directory. Date conversions specified in file dictionaries or in ICONV or OCONV functions use the default date format except where they specifically override it. For information about setting the date format in the *msg.text* file, see Technical Bulletin *UniVerse Internationalization Support* (74-0065).

n Single digit (normally 1 through 4) that specifies the number of digits of the year to output. The default is 4.

* Any single nonnumeric, nonsystem-delimiter character that separates the fields in the case where the conversion must first do a group extraction to obtain the internal date.

m Single digit that must accompany any use of an asterisk. It denotes the number of asterisk-delimited fields to skip in order to extract the date.

s Any single nonnumeric, nonsystem-delimiter character to separate the day, month, and year on output. If you do not specify *s*, the date is converted in 09 SEP 1993 form.

fmt Specifies up to three of the following special format options:

- Y[*n*] Requests only the year number (*n* digits).
- YA Requests only the name of the Chinese calendar year.
- M Requests only the month number (1 through 12).
- MA Requests only the month's name.
- D Requests only the day number within the month (1 through 31).
- W Requests only the day number within the week (1 through 7, where Sunday is 7).
- WA Requests only the day's name.

Q — Requests only the quarter number within the year (1 through 4).

J — Requests only the day number within the year (1 through 366).

[*f1* , *f2* , *f3* **]** — *f1*, *f2*, and *f3* are the format modifiers for the *fmt* options. The brackets are part of the syntax and must be typed. The value of the format modifiers can be any of the following:

n — Specifies how many characters to display.

A[*n*] — Month format is alphabetic. *n* is a number from 1 through 32 specifying how many characters to display.

Z[*n*] — Suppresses leading zeros in day, month, or year. *n* is a number from 1 through 32 specifying how many digits to display.

E — Toggles the European (day / month / year) versus the U.S. (month / day / year) formatting of dates.

L — Specifies that lowercase letters should be retained in month or day names; otherwise the routine converts names to all capitals.

The following table shows which format options can be used together:

Format Option	Can Be Used with These Options
Y	M, MA, D, J, **[***f1* , *f2* , *f3***]**
YA	M, MA, D, **[***f1* , *f2* , *f3***]**
M	Y, YA, D, **[***f1* , *f2* , *f3***]**
MA	Y, YA, D, **[***f1* , *f2* , *f3***]**
D	Y, M, **[***f1* , *f2* , *f3***]**
W	Y, YA, M, MA, D
WA	Y, YA, M, MA, D
Q	
J	Y, **[***f1* , *f2* , *f3***]**

D code: Date Conversion

Each format modifier must correspond to a format option. The following table shows which modifiers can modify which options:

	D	**M**	**Y**	**J**
A	no	yes	no	no
n	yes	yes	yes	yes
Z	yes	yes	yes	yes

The syntax for converting dates with ICONV is the same as for OCONV, except that:

n Ignored. The input conversion accepts any number of year's digits regardless of the *n* specification. If no year exists in the input date, the routine uses the year part of the system date.

s Ignored. The input conversion accepts any single nonnumeric, nonsystem-delimiter character separating the day, month, and year regardless of the *s* specification. If the date input as an undelimited string of characters, it is interpreted as one of the following formats: [YY]YYMMDD or [YY]YYDDD.

subcodes Ignored. The input conversion accepts any combination of upper- and lowercase letters in the month part of the date.

In IDEAL and INFORMATION flavor accounts, the input conversion of an improper date returns a valid internal date and a STATUS( ) value of 3. For example, 02/29/93 is interpreted as 03/01/93, and 09/31/93 is interpreted as 10/01/93. A status of 3 usually represents a common human error. More flagrant errors return an empty string and a STATUS( ) value of 1.

In PICK, REALITY, and IN2 flavor accounts, the input conversion of an improper date always returns an empty string and a status of 1.

If the data to be converted is the null value, a STATUS( ) value of 3 is set and no conversion occurs.

Example

The following example shows how to use the format modifiers:

```
D DMY[Z,A3,Z2]
```

Z modifies the day format option (D) by suppressing leading zeros (05 becomes 5). A3 modifies the month format option (M) so that the month is represented by the first three alphabetic characters (APRIL becomes APR). Z2 modifies the year format option (Y) by suppressing leading zeros and displaying two digits. This conversion converts April 5, 1993 to 5 APR 93.

ECS code: Extended Character Set Conversion

Format

ECS

The ECS code resolves clashes between the UniVerse system delimiters and the ASCII characters CHAR(251) through CHAR(255). It converts the system delimiters and ASCII characters to alternative characters using an appropriate localization procedure. If no localization library is in use, the input string is returned without character conversion. This code is used with an ICONV or OCONV function.

Format

F [;] *element* [; *element* ...]

The F code performs mathematical operations on the data values of a record, or manipulates strings. It comprises any number of operands or operators in reverse Polish format (Lukasiewicz) separated by semicolons.

The program parses the F code from left to right, building a stack of operands. Whenever it encounters an operator, it performs the requested operation, puts the result on the top of the stack, and pops the lower stack elements as necessary.

The semicolon (;) is the element separator.

element can be one or more of the items from the following categories:

A data location or string

loc[R]	Numeric location specifying a data value to be pushed onto the stack, optionally followed by an R (repeat code).
C*n*	*n* is a constant to be pushed onto the stack.
string	Literal string enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ).
number	Constant number enclosed in pairs of double quotation marks ("), single quotation marks ('), or backslashes (\ ). Any integer, positive, negative, or zero can be specified.
D	System date (in internal format).
T	System time (in internal format).

A special system counter operand

@NI	Current item counter (number of items listed or selected).
@ND	Number of detail lines since the last BREAK on a break line.
@NV	Current value counter for columnar listing only.
@NS	Current subvalue counter for columnar listing only.
@NB	Current BREAK level number. 1 = lowest level break. This has a value of 255 on the grand-total line.
@LPV	**L**oad **P**revious **V**alue: load the result of the last correlative code onto the stack.

An operator

Operators specify an operation to be performed on top stack entries. *stack1* refers to the value on the top of the stack, *stack2* refers to the value just below it, *stack3* refers to the value below *stack2*, and so on.

*[*n*]	Multiply *stack1* by *stack2*. The optional *n* is the descaling factor (that is, the result is divided by 10 raised to the *n*th power).
/	Divide *stack1* into *stack2*, result to *stack1*.
R	Same as /, but instead of the quotient, the remainder is returned to the top of the stack.
+	Add *stack1* to *stack2*.
–	Subtract *stack1* from *stack2*, result to *stack1* (except for REALITY flavor, which subtracts *stack2* from *stack1*).
:	Concatenate *stack1* string onto the end of *stack2* string.
[ ]	Extract substring. *stack3* string is extracted, starting at the character specified by *stack2* and continuing for the number of characters specified in *stack1*. This is equivalent to the BASIC [*m*,*n*] operator, where *m* is in *stack2* and *n* is in *stack1*.
S	Sum of multivalues in *stack1* is placed at the top of the stack.
_	Exchange *stack1* and *stack2* values.
P or \	Push *stack1* back onto the stack (that is, duplicate *stack1*).
^	Pop the *stack1* value off the stack.
(*conv*)	Standard conversion operator converts data in *stack1*, putting the result into *stack1*.

A logical operator

Logical operators compare *stack1* to *stack2*. Each returns 1 for true and 0 for false:

=	Equal to.
<	Less than.
>	Greater than.
# or <>	Not equal to.
[	Less than or equal to.
]	Greater than or equal to.

&	Logical AND.
!	Logical OR.
\ *n* \	A label is defined by a positive integer enclosed by two backslashes (\ \).
#*n*	Connection to label *n* if *stack1* differs from *stack2*.
>*n*	Connection to label *n* if *stack1* is greater than *stack2*.
<*n*	Connection to label *n* if *stack1* is less than *stack2*.
=*n*	Connection to label *n* if *stack1* equals *stack2*.
}*n*	Connection to label *n* if *stack1* is greater than or equal to *stack2*.
{*n*	Connection to label *n* if *stack1* is less than or equal to *stack2*.
IN	Tests *stack1* to see if it is the null value.
F*nnnn*	If *stack1* evaluates to false, branch forward *nnnn* characters in the F code, and continue processing.
B*nnnn*	Branch forward unconditionally *nnnn* characters in the F code, and continue processing.
G*nnnn*	Go to label *nnnn*. The label must be a string delimited by backslashes (\ ).
G*	Go to the label defined in *stack1*. The label must be a string delimited by backslashes (\ ).

Note: The F code performs only integer arithmetic.

G code: Group Extraction

Format

G [*skip*] *delim #fields*

The G code extracts one or more values, separated by the specified delimiter, from a field.

skip specifies the number of fields to skip; if it is not specified, 0 is assumed and no fields are skipped.

delim is any single nonnumeric character (except IM, FM, VM, SM, and TM) used as the field separator.

#fields is the decimal number of contiguous delimited values to extract.

Format

L [*n* [,*m*]]

The L code places length constraints on the data to be returned.

If L*n* is specified, selection is met if the value's length is less than or equal to *n* characters; otherwise an empty string is returned.

If L*n*,*m* is specified, selection is met if the value's length is greater than or equal to *n* characters, and less than or equal to *m* characters; otherwise an empty string is returned.

If *n* is omitted or 0, the length of the value is returned.

MC codes: Masked Character Conversion

Formats

MCA
MC/A
MCD[X]
MCL
MCN
MC/N
MCP
MCT
MCU
MCX[D]

The MC codes let you change a field's data to upper- or lowercase, to extract certain classes of characters, to capitalize words in the field, and to change unprintable characters to periods.

MCA	Extracts all alphabetic characters in the field, both upper- and lowercase. Nonalphabetic characters are not printed.
MC/A	Extracts all nonalphabetic characters in the field. Alphabetic characters are not printed.
MCD[X]	Converts decimal to hexadecimal equivalents.
MCL	Converts all uppercase letters to lowercase. Does not affect lowercase letters or nonalphabetic characters.
MCN	Extracts all numeric characters in the field. Alphabetic characters are not printed.
MC/N	Extracts all nonnumeric characters in the field. Numeric characters are not printed.
MCP	Converts each unprintable character to a period.
MCT	Capitalizes the first letter of each word in the field (remainder of word is converted to lowercase).
MCU	Converts all lowercase letters to uppercase. Does not affect uppercase letters or nonalphabetic characters.
MCX[D]	Converts hexadecimal to decimal equivalents.

Format

MD *n* [*m*] [,] [$] [F] [*intl*] [– | < | C | D] [P] [Z] [T] [*fx*]

The MD code converts numeric input data to a format appropriate for internal storage. When converting internal representation of data to external output format, masked decimal conversion inserts the decimal point and other appropriate formats into the data.

The MD must appear in either an ICONV oran OCONV expression. The *n* expression is required and indicates with a single digit the number of decimal places (0 through 9) to be maintained or output. If the value of *n* is 0, the decimal point does not appear in the output.

The optional *m* specifies the power of 10 used to scale the input or output data. On input, the decimal point is moved *m* places to the right before storing. On output, the decimal point is moved *m* places to the left. For example, if *m* is 2 in an input conversion and the input data is 123, it would be stored as 12300. If *m* is 2 in an output conversion and the stored data is 123, it would be output as 1.23. If *m* is not specified, it is assumed to be the same as *n*. In both cases, the last required decimal place is rounded off before excess digits are truncated. Zeros are added if not enough decimal places exist in the original expression.

The remaining specifiers, with the exception of the *fx* option, are meaningful only in the OCONV function; they are ignored when used in the ICONV function.

, Specifies that commas are to be inserted every three digits to the left of the decimal point on output.

$ Prefixes a dollar sign ($) to the number before justification.

F Prefixes a franc sign (F) to the number before justification. (In all flavors except IN2, you must specify F in the conversion code if you want ICONV to accept the character F as a franc sign.)

intl An expression that customizes numeric output according to different international conventions. The *intl* expression can specify a prefix, a suffix, and the characters to use as a thousands delimiter and as the decimal delimiter. The *intl* expression has the following syntax:

[*prefix* **,** *thousands* **,** *decimal* **,** *suffix*]

The bold brackets are part of the syntax and must be typed. The four elements are positional parameters and must be separated by commas. Each element is optional, but its position must be held by a comma. For example, to specify a suffix only, type [`,,,suffix`].

prefix Character string to prefix to the number. If *prefix* contains spaces, commas, or right square brackets, enclose it in quotation marks.

thousands Character string that separates thousands.

decimal Character string to use as a decimal delimiter.

suffix Character string to append to the number. If *suffix* contains spaces, commas, or right square brackets, enclose it in quotation marks.

– Specifies that negative data be suffixed with a minus sign and positive data be suffixed with a blank space.

< Specifies that negative data be enclosed in angle brackets for output; positive data is prefixed and suffixed with a blank space.

C Specifies that negative data include a suffixed CR; positive data is suffixed with two blank spaces.

D Specifies that negative data include a suffixed DB; positive data is suffixed with two blank spaces.

P Specifies that no scaling be performed if the input data already contains a decimal point.

Z Specifies that 0 be output as an empty string.

T Specifies that the data be truncated without rounding.

When converting data to internal format, the *fx* option has the following effect. If the input data has been overlaid on a background field of characters (for example, $###987.65), the *fx* option is used with ICONV to indicate that the background characters should be ignored during conversion. The *f* is a one- or two-digit number indicating the maximum number of background characters to be ignored. The *x* specifies the background character to be ignored. If background characters exist in the input data and you do not use the *fx* option, the data is considered bad and an empty string results.

When converting data from internal representation to external output format, the *fx* option causes the external output of the data to overlay a field of background

characters. The *f* is a one- or two-digit number indicating the number of times the background character is to be repeated. The *x* specifies the character to be used as a background character. If the $ option is used with the *fx* option, the $ precedes the background characters when the data is output.

ML and MR codes: Formatting Numbers

Format

ML [*n* [*m*] [Z] [,] [C | D | M | E | N] [$] [*intl*] [(*fx*)]
MR [*n* [*m*] [Z] [,] [C | D | M | E | N] [$] [*intl*] [(*fx*)]

The ML and MR codes allow special processing and formatting of numbers and dollar amounts. ML specifies left justification; MR specifies right justification.

n Number of digits to be printed to the right of the decimal point. If *n* is omitted or 0, no decimal point is printed.

m Descales (divided) the number on file by 10 to the *m*th power. If not specified, *m* = *n* is assumed. On input, the decimal point is moved *m* places to the right before storing. On output, the decimal point is moved *m* places to the left. For example, if *m* is 2 in an input conversion specification and the input data is 123, it would be stored as 12300. If *m* is 2 in an output conversion specification and the stored data is 123, it would be output as 1.23. If the *m* is not specified, it is assumed to be the same as the *n* value. In both cases, the last required decimal place is rounded off before excess digits are truncated. Zeros are added if not enough decimal places exist in the original expression.

The remaining specifiers, with the exception of the *fx* option, are meaningful only in the OCONV function; they are ignored when used in the ICONV function.

Z Specifies that 0 be output as an empty string.

, Specifies that commas be inserted every three digits to the left of the decimal point on output.

C Suffixes negative values with CR.

D Suffixes positive values with DB.

M Suffixes negative numbers with a minus sign (–).

E Encloses negative numbers in angle brackets (< >).

N Suppresses the minus sign (–) on negative numbers.

$ Prefixes a dollar sign ($) to the number before justification. The $ option automatically justifies the number and places the $ just before the first digit of the number output.

intl An expression that customizes numeric output according to different international conventions. The *intl* expression can specify a prefix, a suffix, and the characters to use as a thousands delimiter and as the decimal delimiter. The *intl* expression has the following syntax:

[*prefix* **,** *thousands* **,** *decimal* **,** *suffix* **]**

The bold brackets are part of the syntax and must be typed. The four elements are positional parameters and must be separated by commas. Each element is optional, but its position must be held by a comma. For example, to specify a suffix only, type [`,,,suffix`].

prefix Character string to prefix to the number. If *prefix* contains spaces, commas, or right square brackets, enclose it in quotation marks.

thousands Character string that separates thousands.

decimal Character string to use as a decimal delimiter.

suffix Character string to append to the number. If *suffix* contains spaces, commas, or right square brackets, enclose it in quotation marks.

f One of three format codes:

Data justifies in a field of *x* blanks.

* Data justifies in a field of *x* asterisks (*).

% Data justifies in a field of *x* 0.

The format codes are followed by *x*, the number that specifies the size of the field.

Literal strings can also be enclosed in the parentheses. The text is printed as specified, with the number being processed right- or left-justified.

MP code: Packed Decimal Conversion

Format

MP

The MP code allows decimal numbers to be packed two-to-the-byte for storage. Packed decimal numbers occupy approximately half the disk storage space required by unpacked decimal numbers.

Leading + signs are ignored. Leading – signs cause a hexadecimal D to be stored in the lower half of the last internal digit. If there is an odd number of packed halves, four leading bits of 0 are added. The range of the data bytes in internal format expressed in hexadecimal is 00 through 99 and 0D through 9D. Only valid decimal digits (0 – 9) and signs (+, –) should be input. Other characters cause no conversion to take place.

Packed decimal numbers should always be unpacked for output, since packed values that are output unconverted are not displayed on terminals in a recognizable format.

Format

MT [H] [S] [*c*]

The MT code converts times from conventional formats to an internal format for storage. It also converts internal times back to conventional formats for output. When converting input data to internal storage format, time conversion specifies the format that is to be used to enter the time. When converting internal representation of data to external output format, time conversion defines the external output format for the time.

MT is required when you specify time in either the ICONV or the OCONV function. The remaining specifiers are meaningful only in the OCONV function; they are ignored when used in the ICONV function.

The internal representation of time is the numeric value of the number of seconds since midnight.

If used with ICONV in an IDEAL, INFORMATION, or PIOPEN flavor account, the value of midnight is 0. In all other account flavors, the value of midnight is 86400.

You can use any nonnumeric character to separate the hours, minutes, and seconds. If no minutes or seconds are entered, they are assumed to be 0. You can use a suffix of AM, A, PM, or P to specify that the time is before or after noon. If an hour larger than 12 is entered, a 24-hour clock is assumed. (Note: 12:00 AM is midnight and 12:00 PM is noon.)

H Specifies that a 12-hour format with the suffixes AM or PM is to be output. The default format is the 24-hour format.

S Specifies that seconds are to be output. The default omits seconds.

c Specifies the character to be used to separate the hours, minutes, and seconds in the output. The default character is a colon (:).

MX, MO, and MB codes: Radix Conversion

Formats

MX [0C]	Hexadecimal conversion (base 16)
MO [0C]	Octal conversion (base 8)
MB [0C]	Binary conversion (base 2)

The MX, MO, and MB codes listed in this section control the conversion of data represented in hexadecimal, octal, and binary format into decimal (base 10) format when used with the ICONV function. The decimal, or ASCII, format is the internal format for data representation.

The appropriate specification (MX, MO, or MB) is used according to whether the data to be converted is represented in hexadecimal, octal, or binary format respectively.

Characters outside of the range for each of the bases produce conversion errors. The ranges are as follows:

MX (hexadecimal)	0 through 9, A through F, a through f
MO (octal)	0 through 7
MB (binary)	0, 1

When used alone, the MX, MO, and MB specifications produce decimal values. When the 0C option is used with any of the three previous specifications, the equivalent ASCII character is generated rather than the decimal value.

When used with the OCONV function, the MX, MO, and MB specifications control the conversion of numeric and alphanumeric data from their internal representation to hexadecimal, octal, or binary format for output.

The appropriate specification (MX, MO, or MB) is used according to whether the data is to be converted to hexadecimal, octal, or binary format. Nonnumeric data produces a conversion error if the 0C option is not used.

When an MX, MO, or MB specification is used alone, a decimal value is assumed. When the 0C option is used with any of the three previous specifications, the data is assumed to be an ASCII character (or character string); each character in the string is converted to the hexadecimal, octal, or binary format of its ASCII character code. Data is converted from left to right, one character at a time, until all data is exhausted.

Format

MY

The MY code specifies conversion from hexadecimal to ASCII on *output*, and ASCII to hexadecimal on *input*.

Characters outside of the range for each of the bases produce conversion errors. The ranges are as follows:

MY (hexadecimal)	0 through 9, A through F, a through f

When used with the OCONV function, MY converts from hexadecimal to ASCII. When used with the ICONV function, MY converts from ASCII to hexadecimal.

NR code: Roman Numeral Conversion

Format

NR

The NR code converts Roman numerals into Arabic numerals when used with the ICONV function. The decimal, or ASCII, format is the internal format for representation.

When used with the OCONV function, the NR code converts Arabic numerals into Roman numerals.

The following is a table of Roman / Arabic numeral equivalents:

Roman	Arabic
i	1
v	5
x	10
l	50
c	100
d	500
m	1000
V	5000
X	10,000
L	50,000
C	100,000
D	500,000
M	1,000,000

Format

P(*pattern*) [{ ; | / } (*pattern*) ...]

The P code extracts data whose values match one or more patterns. If the data does not match any of the patterns, an empty string is returned.

pattern can contain one or more of the following codes:

*n*N	An integer followed by the letter N, which tests for *n* numeric characters.
*n*A	An integer followed by the letter A, which tests for *n* alphabetic characters.
*n*X	An integer followed by the letter X, which tests for *n* alphanumeric characters.
nnnn	A literal string, which tests for that literal string.

If *n* is 0, any number of numeric, alphabetic, or alphanumeric characters matches. If either the data or the match pattern is the null value, null is returned.

Separate multiple ranges by a semicolon (;) or a slash (/).

Parentheses must enclose each pattern to be matched. For example, if the user wanted only Social Security numbers returned, then P(3N-2N-4N) would test for strings of exactly three numbers, then a hyphen, then exactly two numbers, then a hyphen, then exactly four numbers.

Q code: Exponential Notation

Format

Q [R | L | X] [*n* { E | . } *m*] [*edit*] [*mask*]

The Q code converts numeric input data from exponential notation to a format appropriate for internal storage. When converting internal representation of data to external output format, the Q code converts the data to exponential notation by determining how many places to the right of the decimal point are to be displayed and by specifying the exponent.

Q alone and QR both specify right justification. QL specifies left justification.

QX must be used alone. X is not valid with any additional qualifiers such as *n*, E*m*, *edit*, or *mask*.

n specifies the number of fractional digits to the right of the decimal point. It can be a number from 0 through 9.

m specifies the exponent. It can be a number from 0 through 9. When used with E, *m* can also be a negative number from –1 through –9.

Separate *n* and *m* with either the letter E or a period (.). Use E if you want to specify a negative exponent.

edit can be any of the following:

- $ Prefixes a dollar sign to the value.
- F Prefixes a franc sign to the value.
- , Inserts commas after every thousand.
- Z Returns an empty string if the value is 0. Any trailing fractional zeros are suppressed, and a zero exponent is suppressed.
- E Surrounds negative numbers with angle brackets (< >).
- C Appends cr to negative numbers.
- D Appends db to positive numbers.
- B Appends db to negative numbers.
- N Suppresses a minus sign on negative numbers.
- M Appends a minus sign to negative numbers.
- X Specifies right justification. QX must be used alone. X is not valid with any other code. QX is synonymous with QR0E0 as input and MR as output.
- T Truncates instead of rounding.

mask allows literals to be intermixed with numerics in the formatted output field. The mask can include any combination of literals and the following three special format mask characters:

#*n*	Data is displayed in a field of *n* fill characters. A blank is the default fill character. It is used if the format string does not specify a fill character after the width parameter.
%*n*	Data is displayed in a field of *n* zeros.
***n*	Data is displayed in a field of *n* asterisks.

See the FMT function for more information about formatting numbers.

R code: Range Function

Format

R*n,m* [{ ; | / } *n,m* ...]

The R code limits returned data to that which falls within specified ranges. *n* is the lower bound, *m* is the upper bound.

Separate multiple ranges by a semicolon (;) or a slash (/).

If range specifications are not met, an empty string is returned.

S (Soundex) code

Format

S

The S code with no arguments specifies a soundex conversion. Soundex is a phonetic converter. It converts ordinary English words into a four-character abbreviation consisting of one alphabetic character followed by three digits. Soundex conversions are frequently used to build indices for name lookups.

S (substitution) code

Format

S ; *nonzero.substitute* ; *zero.substitute* ; *null.substitute*

The S code substitutes one of three values depending on whether the data value evaluates to zero or an empty string, to the null value, or to something else.

If the data being converted evaluates to zero or an empty string, *zero.substitute* is returned. If the data is nonzero, nonempty, and nonnull, *nonzero.substitute* is returned. If the data is the null value, *null.substitute* is returned. If *null.substitute* is omitted, null values are not replaced.

All three substitute expressions can be either a quoted string, a field number, or an asterisk. If it is an asterisk and the data evaluates to something other than zero, the empty string, or the null value, the data value itself is returned.

Example

Assume a BASIC program where @RECORD is:

```
AFBFCvD
```

Statement	Output
`PRINT OCONV("x","S;2;'zero'")`	`B`
`PRINT OCONV("x","S;*;'zero'")`	`x`
`PRINT OCONV(0,"S;2;'zero'")`	`zero`
`PRINT OCONV('',"S;*;'zero'")`	`zero`

Format

T [*start*,] *length*

The T code extracts a contiguous string of characters from a field.

start Starting column number. If you omit it, 1 is assumed.

length Number of characters to extract.

If *start* is not specified, the extraction is either from the left or from the right depending on the justification specified in line 5 of the dictionary definition item. If, in a BASIC program, *start* is not specified, the extraction is from the right. In this case the starting position is calculated according to the following formula:

string.length – *substring.length* + 1

This lets you extract the last *n* characters of a string without having to calculate the string length.

If *start* is specified, extraction is always from left to right.

Tfile code: File Translation

Format

T[DICT] *filename* ; *c* [*vloc*] ; [*iloc*] ; [*oloc*] [;*bloc*]

T[DICT] *filename* ; *c* ; [*iloc*] ; [*oloc*] [;*bloc*] [,*vloc* | **[***vloc***]**]

The T*file* code converts values from one file to another by translating through a file. It uses data values in the source file as IDs for records in a lookup file. The source file can then reference values in the lookup file.

To access the lookup file, its record IDs (field 0) must be referenced. If no reference is made to the record IDs of the lookup file, the file cannot be opened and the conversion cannot be performed. The data value being converted must be a record ID in the lookup file.

DICT — Specifies the lookup file's dictionary. (In REALITY flavor accounts, you can use an asterisk (*) to specify the dictionary: for instance, T**filename* … .)

filename — Name of the lookup file.

c — Translation subcode, which must be one of the following:

- V Conversion item must exist on file, and the specified field must have a value, otherwise an error message is returned.
- C If conversion is impossible, return the original value-to-be-translated.
- I Input verify only. Functions like V for input and like C for output.
- N Returns the original value-to-be-translated if the null value is found.
- O Output verify only. Functions like C for input and like V for output.
- X If conversion is impossible, return an empty string.

vloc — Number of the value to be returned from a multivalued field. If you do not specify *vloc* and the field is multivalued, the whole field is returned with all system delimiters turned into blanks. If the *vloc* specification follows the *oloc* or *bloc* specification, enclose *vloc* in square brackets or separate *vloc* from *oloc* or *bloc* with a comma.

iloc Field number (decimal) for *input* conversion. The input value is used as a record ID in the lookup file, and the translated value is retrieved from the field specified by the *iloc*. If the *iloc* is omitted, no input translation takes place.

oloc Field number (decimal) for *output* translation. When RetrieVe creates a listing, data from the field specified by *oloc* in the lookup file are listed instead of the original value.

bloc Field number (decimal) which is used instead of *oloc* during the listing of BREAK.ON and TOTAL lines.

D

BASIC Reserved Words

This appendix lists reserved words in the BASIC language. We recommend that you not use them as variable names in your programs.

ABORT
ABORTE
ABORTM
ABS
ABSS
ACOS
ADDS
ALL
ALPHA
AND
ANDS
ARG.
ASCII
ASIN
ASSIGN
ASSIGNED
ATAN
AUTHORIZATION
BCONVERT
BEFORE
BEGIN
BITAND
BITNOT
BITOR
BITRESET
BITSET
BITTEST
BITXOR
BREAK
BSCAN
BY
CALL
CALLING
CAPTURING
CASE
CAT
CATS
CHAIN
CHANGE
CHAR
CHARS
CHECKSUM
CLEAR
CLEARCOMMON
CLEARDATA
CLEARFILE
CLEARINPUT
CLEARPROMPTS
CLEARSELECT
CLOSE
CLOSESEQ
COL1
COL2
COM
COMMIT
COMMON
COMPARE
CONTINUE
CONVERT
COS
COSH
COUNT
COUNTS
CREATE
CRT
DATA
DATE
DCOUNT
DEBUG
DECLARE
DEFFUN
DEL
DELETE
DELETELIST
DELETEU
DIAGNOSTICS
DIM
DIMENSION
DISPLAY
DIV
DIVS
DO
DOWNCASE
DQUOTE
DTX
EBCDIC
ECHO
ELSE
END
ENTER
EOF
EQ
EQS
EQU
EQUATE
EREPLACE
ERRMSG
ERROR
EXCHANGE
EXEC
EXECUTE
EXIT
EXP
EXTRACT
FADD
FDIV
FFIX
FFLT
FIELD
FIELDS
FIELDSTORE
FILEINFO
FILELOCK
FILEUNLOCK
FIND
FINDSTR
FIX
FLUSH
FMT
FMTS
FMUL
FOLD
FOOTING
FOR
FORMLIST
FROM
FSUB
FUNCTION
GARBAGECOLLECT
GCI
GE
GES
GET
GETLIST
GETREM

GETX
GO
GOSUB
GOTO
GROUP
GROUPSTORE
GT
GTS
HEADING
HEADINGE
HEADINGN
HUSH
ICHECK
ICONV
ICONVS
IF
IFS
ILPROMPT
IN
INCLUDE
INDEX
INDEXS
INDICES
INMAT
INPUT
INPUTCLEAR
INPUTDISP
INPUTERR
INPUTIF
INPUTNULL
INPUTTRAP
INS
INSERT
INT
ISNULL
ISNULLS
ISOLATION
ITYPE
KEY
KEYEDIT
KEYEXIT
KEYIN
KEYTRAP
LE
LEFT
LEN
LENS
LES
LET
LEVEL
LIT
LITERALLY
LN
LOCATE
LOCK
LOCKED
LOOP
LOWER
LPTR
LT
LTS
MAT
MATBUILD
MATCH
MATCHES
MATCHFIELD
MATPARSE
MATREAD
MATREADL
MATREADU
MATWRITE
MATWRITEU
MAXIMUM
MESSAGE
MINIMUM
MOD
MODS
MTU
MULS
NAP
NE
NEG
NEGS
NES
NEXT
NOBUF
NO.ISOLATION
NOT
NOTS
NULL
NUM
NUMS
OCONV
OCONVS
OFF
ON
OPEN
OPENCHECK
OPENDEV
OPENPATH
OPENSEQ
OR
ORS
OUT
PAGE
PASSLIST
PCDRIVER
PERFORM
PRECISION
PRINT
PRINTER
PRINTERIO
PRINTERR
PROCREAD
PROCWRITE
PROG
PROGRAM
PROMPT
PWR
QUOTE
RAISE
RANDOMIZE
READ
READ.COMMITTED
READ.UNCOM-
MITTED
READBLK
READL
READLIST
READNEXT
READSEQ
READT
READU
READV
READVL
READVU
REAL
RECIO
RECORDLOCKED
RECORDLOCKL
RECORDLOCKU
RELEASE
REM
REMOVE
REPEAT
REPEATABLE.READ
REPLACE
RESET
RETURN
RETURNING
REUSE
REVREMOVE
REWIND
RIGHT
RND
ROLLBACK
RPC.CALL
RPC.CONNECT
RPC.DISCONNECT
RQM
RTNLIST
SADD
SCMP
SDIV
SEEK
SELECT
SELECTE
SELECTINDEX
SELECTN
SELECTV
SEND
SENTENCE
SEQ
SEQS
SEQSUM
SERIALIZABLE
SET
SETREM
SETTING
SIN
SINH
SLEEP
SMUL
SOUNDEX
SPACE
SPACES
SPLICE
SQLALLOCONNECT
SQLALLOCENV
SQLALLOCSTMT
SQLBINDCOL
SQLCANCEL
SQLCOLATTRI-
BUTES
SQLCONNECT
SQLDESCRIBECOL
SQLDISCONNECT
SQLERROR
SQLEXECDIRECT
SQLEXECUTE

SQLFETCH
SQLFREECONNECT
SQLFREEENV
SQLFREESTMT
SQLGETCURSOR-
NAME
SQLNUMRESULT-
COLS
SQLPREPARE
SQLROWCOUNT
SQLSETCONNECT-
OPTION
SQLSETCURSOR-
NAME
SQLSETPARAM
SQRT
SQUOTE
SSELECT
SSELECTN
SSELECTV
SSUB
START
STATUS
STEP
STOP
STOPE
STOPM
STORAGE
STR
STRS
SUB
SUBR
SUBROUTINE
SUBS
SUBSTRINGS
SUM
SUMMATION
SYSTEM
TABSTOP
TAN
TANH
TERMINFO
THEN
TIME
TIMEDATE
TIMEOUT
TO
TPARM
TPRINT
TRANS
TRANSACTION
TRIM
TRIMB
TRIMBS
TRIMF
TRIMFS
TRIMS
TTYCTL
TTYGET
TTYSET
UNASSIGNED
UNIT
UNLOCK
UNTIL
UPCASE
USING
WEOF
WEOFSEQ
WEOFSEQF
WHILE
WORDSIZE
WORKWRITE
WRITEBLK
WRITELIST
WRITESEQ
WRITESEQF
WRITET
WRITEU
WRITEV
WRITEVU
XLATE
XTD

E

@Variables

Table E-1 lists BASIC @variables. The @variables denoted by an asterisk (*) are read-only. All others can be changed by the user.

The EXECUTE statement initializes the values of stacked @variables either to 0 or to values reflecting the new environment. These values are not passed back to the calling environment. The values of nonstacked @variables are shared between the EXECUTE and calling environments. All @variables listed here are stacked unless otherwise indicated.

Table E-1. BASIC @Variables

Variable	Read-only	Value
@ABORT.CODE	*	A numeric value indicating the type of condition that caused the ON.ABORT paragraph to execute. The values are: 1 - An ABORT statement was executed. 2 - An abort was requested after pressing the **Break** key followed by option A. 3 - An internal or fatal error occurred.
@ACCOUNT	*	User login name. Same as @LOGNAME. Nonstacked.
@AM	*	Field mark: CHAR(254). Same as @FM.
@ANS		Last I-type answer, value indeterminate.
@AUTHORIZATION	*	Current effective user name.
@CRTHIGH	*	Number of lines on the terminal.
@CRTWIDE	*	Number of columns on the terminal.
@COMMAND	*	The last command executed or entered at the TCL.

Table E-1. BASIC @Variables (Continued)

Variable	Read-only	Value
@COMMAND.STACK	*	Dynamic array containing the last 99 commands executed.
@CONV		For future use.
@DATA.PENDING	*	Dynamic array containing input generated by the DATA statement. Values in the dynamic array are separated by field marks.
@DATE		Internal date.
@DAY		Day of month from @DATE.
@DICT		For future use.
@FALSE	*	Compiler replaces the value with 0.
@FILENAME		Current filename. Same as @FILE.NAME.
@FILE.NAME		Current filename. Same as @FILENAME.
@FM	*	Field mark: CHAR(254). Same as @AM.
@FORMAT		For future use.
@HDBC	*	ODBC connection environment on the local UniVerse server. Nonstacked.
@HEADER		For future use.
@HENV	*	ODBC environment on the local UniVerse server. Nonstacked.
@HSTMT	*	ODBC statement environment on the local UniVerse server. Nonstacked.
@ID		Current record ID.
@IM	*	Item mark: CHAR(255).
@ISOLATION	*	Current transaction isolation level for the active transaction or the current default isolation level if no transaction exists.
@LEVEL	*	The nesting level of execution statements. Nonstacked.
@LOGNAME	*	User login name. Same as @ACCOUNT.
@LPTRHIGH	*	Number of lines on the device to which you are printing (that is, terminal or printer).

Table E-1. BASIC @Variables (Continued)

Variable	Read-only	Value
@LPTRWIDE	*	Number of columns on the device to which you are printing (that is, terminal or printer).
@MONTH		Current month.
@MV		The current value counter for columnar listing only. Used only in I-descriptors. Same as @NV.
@NB		The current BREAK level number. 1 is the lowest-level break. @NB has a value of 255 on the grand total line. Used only in I-descriptors.
@ND		The number of detail lines since the last BREAK on a break line. Used only in I-descriptors.
@NI		The current item counter (the number of items listed or selected). Used only in I-descriptors. Same as @RECCOUNT.
@NS		The current subvalue counter for columnar listing only. Used only in I-descriptors.
@NULL	*	The null value. Nonstacked.
@NULL.STR	*	The internal representation of the null value, which is CHAR(128). Nonstacked.
@NV		The current value counter for columnar listing only. Used only in I-descriptors. Same as @MV.
@OPTION		The value of field 5 in the VOC for the calling verb.
@PARASENTENCE	*	The last sentence or paragraph that invoked the current process.
@PATH	*	Pathname of the current account.
@RECCOUNT		The current item counter (the number of items listed or selected). Used only in I-descriptors. Same as @NI.
@RECORD		Entire current record.
@RECUR0		Reserved.
@RECUR1		Reserved.

Table E-1. BASIC @Variables (Continued)

Variable	Read-only	Value
@RECUR2		Reserved.
@RECUR3		Reserved.
@RECUR4		Reserved.
@SCHEMA	*	Schema name of the current UniVerse account. Nonstacked. When users create a new schema, @SCHEMA is not set until the next time they log in to UniVerse.
@SELECTED		Number of elements selected from the last select list. Nonstacked.
@SENTENCE	*	The sentence that invoked the current BASIC program. Any EXECUTE updates @SENTENCE.
@SM	*	Subvalue mark: CHAR(252). Same as @SVM.
@SQL.CODE	*	For future use.
@SQL.ERROR	*	For future use.
@SQL.STATE	*	For future use.
@SQL.WARNING	*	For future use.
@SQLPROC.NAME	*	The name of the current SQL procedure.
@SQLPROC.TX.LEVEL	*	The transaction level at which the current SQL procedure began.
@STDFIL		Default file variable.
@SVM	*	Subvalue mark: CHAR(252). Same as @SM.
@SYS.BELL	*	Bell character. Nonstacked.
@SYSTEM.RETURN.CODE		Status codes returned by system processes. Same as @SYSTEM.SET.
@SYSTEM.SET		Status codes returned by system processes. Same as @SYSTEM.RETURN.CODE.
@TERM.TYPE	*	The terminal type. Nonstacked.
@TIME		Internal time.
@TM	*	Text mark: CHAR(251).

Table E-1. BASIC @Variables (Continued)

Variable	Read-only	Value
@TRANSACTION	*	A numeric value. Any nonzero value indicates that a transaction is active; the value 0 indicates that no transaction exists.
@TRANSACTION.ID	*	Transaction number of the active transaction. An empty string indicates that no transaction exists.
@TRANSACTION.LEVEL	*	Transaction nesting level of the active transaction. A 0 indicates that no transaction exists.
@TRUE		Compiler replaces the value with 1.
@TTY		Terminal device name. If the process is a phantom, @TTY returns the value 'phantom'. If the process is a UniVerse API, it returns 'uvcs'.
@USER0		User-defined.
@USER1		User-defined.
@USER2		User-defined.
@USER3		User-defined.
@USER4		User-defined.
@USERNO	*	User number. Nonstacked. Same as @USER.NO.
@USER.NO	*	User number. Nonstacked. Same as @USERNO.
@USER.RETURN.CODE		Status codes created by the user.
@VM	*	Value mark CHAR(253).
@WHO	*	The name of the current UniVerse account directory. Nonstacked.
@YEAR		Current year.

F

BASIC Subroutines

This appendix describes the following subroutines you can call from a UniVerse BASIC program:

!ASYNC (!AMLC)

!ERRNO

!FCMP

!GET.PARTNUM

!GET.PATHNAME

!GETPU

!GET.USER.COUNTS

!INLINE.PROMPTS

!INTS

!MAKE.PATHNAME

!MATCHES

!MESSAGE

!REPORT.ERROR

!SET.PTR

!SETPU

!TIMDAT

!USER.TYPE

!VOC.PATHNAME

In addition, the subroutines listed in Table F-1 have been added to existing functions for PI/open compatibility.

Table F-1. PI/open Subroutines

Subroutine	Associated Function
CALL !ADDS	ADDS
CALL !ANDS	ANDS
CALL !CATS	CATS
CALL !CHARS	CHARS
CALL !CLEAR.PROMPTS	CLEAR.PROMPTS
CALL !COUNTS	COUNTS
CALL !DIVS	DIVS
CALL !EQS	EQS
CALL !FADD	FADD
CALL !FDIV	FDIV
CALL !FIELDS	FIELDS
CALL !FMTS	FMTS
CALL !FMUL	FMUL
CALL !FOLD	FOLD
CALL !FSUB	FSUB
CALL !GES	GES
CALL !GTS	GTS
CALL !ICONVS	ICONVS
CALL !IFS	IFS
CALL !INDEXS	INDEXS
CALL !LENS	LENS
CALL !LES	LES
CALL !LTS	LTS
CALL !MAXIMUM	MAXIMUM
CALL !MINIMUM	MINIMUM
CALL !MODS	MODS
CALL !MULS	MULS

Table F-1. PI/open Subroutines (Continued)

Subroutine	Associated Function
CALL !NES	NES
CALL !NOTS	NOTS
CALL !NUMS	NUMS
CALL !OCONVS	OCONVS
CALL !ORS	ORS
CALL !SEQS	SEQS
CALL !SPACES	SPACES
CALL !SPLICE	SPLICE
CALL !STRS	STRS
CALL !SUBS	SUBS
CALL !SUBSTRINGS	SUBSTRINGS
CALL !SUMMATION	SUMMATION

!ASYNC subroutine

Syntax

CALL !ASYNC (*key*, *line*, *data*, *count*, *carrier*)

Description

Use the !ASYNC subroutine (or its synonym !AMLC) to send data to, and receive data from an asynchronous device.

key defines the action to be taken (1 through 5). The values for *key* are defined in the following list:

line is the number portion from the &DEVICE& entry TTY##, where ## represents a decimal number.

data is the data being sent to or received from the line.

count is an output variable containing the character count.

carrier is an output variable that returns a value dependent on the value of *key*. If *key* is 1, 2, or 3, *carrier* returns the variable specified by the user. If *key* has a value of 4 or 5, *carrier* returns 1.

You must first assign an asynchronous device using the ASSIGN command. A entry must be in the &DEVICE& file for the device to be assigned with the record ID format of TTY##, where ## represents a decimal number. The actions associated with each key value are as follows:

key	**Action**
1	Inputs the number of characters indicated by the value of *count*.
2	Inputs the number of characters indicated by the value of *count* or until a linefeed character is encountered.
3	Outputs the number of characters indicated by the value of *count*.
4	Returns the number of characters in the input buffer to *count*. On operating systems where the FIONREAD key is not supported, 0 is returned in *count*. When the value of *key* is 4, 1 is always returned to *carrier*.
5	Returns 0 in *count* if there is insufficient space in the output buffer. On operating systems where the TIOCOUTQ key is not supported, 0 is returned in *count*. When the value of *key* is 5, 1 is always returned to *carrier*.

Example

The !ASYNC subroutine returns the first 80 characters from the device defined by ASYNC10 in the &DEVICE& file to the variable data.

```
data=
count= 80
carrier= 0
call !ASYNC(1,10,data,count,carrier)
```

!ERRNO subroutine

Syntax

CALL !ERRNO (*variable*)

Description

Use the !ERRNO subroutine to return the current value of the operating system *errno* variable.

variable is the name of a BASIC variable.

The !ERRNO subroutine returns the value of the system *errno* variable after the last call to a GCI subroutine in *variable*. If you call a system routine with the GCI, and the system call fails, you can use !ERRNO to determine what caused the failure. If no GCI routine was called prior to its execution, !ERRNO returns 0. The values of *errno* that apply to your system are listed in the system include file *errno.h*.

Syntax

CALL !FCMP (RESULT, X,Y)

Description

Use the !FCMP subroutine to compare the equality of two floating-point numeric values as follows:

```
IF X < Y THEN RESULT = -1
IF X = Y THEN RESULT = 0
IF X > Y THEN RESULT = 1
```

!GET.PARTNUM subroutine

Syntax

CALL !GET.PARTNUM (*file*, *record.ID*, *partnum*, *status*)

Description

Use the !GET.PARTNUM subroutine with distributed files to determine the number of the part file to which a given record ID belongs.

file (input) is the file variable of the open distributed file.

record.ID (input) is the record ID.

partnum (output) is the part number of the part file of the distributed file to which the given record ID maps.

status (output) is 0 for a valid part number or an error number for an invalid part number. An insert file of equate tokens for the error numbers is available.

An insert file of equate names is provided to allow you to use mnemonics for the error numbers. The insert file is called INFO_ERRORS.INS.IBAS, and is located in the *INCLUDE* subdirectory. To use the insert file, specify $INCLUDE SYSCOM INFO_ERRORS.INS.IBAS when you compile the program.

Equate Name	Description
IE$NOT.DISTFILE	The file specified by the file variable is not a distributed file.
IE$DIST.DICT.OPEN.FAIL	The program failed to open the file dictionary for the distributed file.
IE$DIST.ALG.READ.FAIL	The program failed to read the partitioning algorithm from the distributed file dictionary.
IE$NO.MAP.TO.PARTNUM	The record ID specified is not valid for this distributed file.

Use the !GET.PARTNUM subroutine to call the partitioning algorithm associated with a distributed file. If the part number returned by the partitioning algorithm is not valid, that is, not an integer greater than zero, !GET.PARTNUM returns a nonzero status code. If the part number returned by the partitioning algorithm is valid, !GET.PARTNUM returns a zero status code.

Note: !GET.PARTNUM does not check that the returned part number corresponds to one of the available part files of the currently opened file.

Example

In the following example, a distributed file SYS has been defined with parts and part numbers S1, 5, S2, 7, and S3, 3, respectively. The file uses the default SYSTEM partitioning algorithm.

```
PROMPT ''
GET.PARTNUM = '!GET.PARTNUM'
STATUS = 0
PART.NUM = 0
OPEN '', 'SYS' TO FVAR ELSE STOP 'NO OPEN SYS'
PATHNAME.LIST = FILEINFO(FVAR, FINFO$PATHNAME)
PARTNUM.LIST = FILEINFO(FVAR, FINFO$PARTNUM)
LOOP
    PRINT 'ENTER Record ID : ':
    INPUT RECORD.ID
 WHILE RECORD.ID
    CALL @GET.PARTNUM(FVAR, RECORD.ID, PART.NUM, STATUS)
    LOCATE PART.NUM IN PARTNUM.LIST<1> SETTING PART.INDEX THEN
PATHNAME = PATHNAME.LIST <PART.INDEX>
    END ELSE
          PATHNAME = ''
   END
    PRINT 'PART.NUM = ':PART.NUM:' STATUS = ':STATUS :'
PATHNAME = ': PATHNAME
 REPEAT
 END
```

!GET.PARTNUM returns part number 5 for input record ID 5-1, with status code 0, and part number 7 for input record ID 7-1, with status code 0, and part number 3 for input record ID 3-1, with status code 0. These part numbers are valid and correspond to available part files of file SYS.

!GET.PARTNUM returns part number 1200 for input record ID 1200-1, with status code 0. This part number is valid but does not correspond to an available part file of file SYS.

!GET.PARTNUM returns part number 0 for input record ID 5-1, with status code IE$NO.MAP.TO.PARTNUM, and part number 0 for input record ID A-1, with status code IE$NO.MAP.TO.PARTNUM, and part number 0 for input record ID 12-4, with status code IE$NO.MAP.TO.PARTNUM. These part numbers are not valid and do not correspond to available part files of the file SYS.

!GET.PATHNAME subroutine

Syntax

CALL !GET.PATHNAME (*pathname*, *directoryname*, *filename*, *status*)

Description

Use the !GET.PATHNAME subroutine to return the directory name and filename parts of a pathname.

pathname (input) is the pathname from which the details are required.

directoryname (output) is the directory name portion of the pathname, that is, the pathname with the last entry name stripped off.

filename (output) is the filename portion of the pathname.

status (output) is the returned status of the operation. A 0 indicates success, another number is an error code indicating that the supplied pathname was not valid.

Example

If *pathname* is input as */usr/accounts/ledger*, *directoryname* is returned as */usr/accounts*, and *filename* is returned as *ledger*.

```
PATHNAME = "/usr/accounts/ledger "
CALL !GET.PATHNAME(PATHNAME,DIR,FNAME,STATUS)
IF STATUS = 0
THEN
   PRINT "Directory portion = ":DIR
   PRINT "Entryname portion = ":FNAME
END
```

Syntax

CALL !GETPU (*key, print.channel, set.value, return.code*)

Description

Use the !GETPU subroutine to read individual parameters of any logical print channel.

key is a number indicating the parameter to be read.

print.channel is the logical print channel, designated by –1 through 255.

set.value is the value to which the parameter is currently set.

return.code is the code returned.

The !GETPU subroutine allows you to read individual parameters of logical print channels as designated by *print.channel.* Print channel 0 is the terminal unless a PRINTER ON statement has been executed to send output to the default printer. If you specify print channel –1, the output is directed to the terminal, regardless of the status of PRINTER ON or OFF. See the description of the !SETPU subroutine later in this appendix for a means of setting individual *print.channel* parameters.

Equate Names for Keys

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is GETPU.INS.IBAS. Use the $INCLUDE compiler directive to insert this file if you want to use equate names. The following list shows the equate names and keys for the parameters:

Mnemonic	Key	Parameter
PU$MODE	1	Printer mode.
PU$WIDTH	2	Device width (columns).
PU$LENGTH	3	Device length (lines).
PU$TOPMARGIN	4	Top margin (lines).
PU$BOTMARGIN	5	Bottom margin (lines).
PU$LEFTMARGIN	6	Left margin (columns, reset on printer close). Always returns 0.
PU$SPOOLFLAGS	7	Spool option flags.

Mnemonic	Key	Parameter
PU$DEFERTIME	8	Spool defer time. This cannot be 0.
PU$FORM	9	Spool form (string).
PU$BANNER	10	Spool banner or hold filename (string).
PU$LOCATION	11	Spool location (string).
PU$COPIES	12	Spool copies. A single copy can be returned as 1 or 0.
PU$PAGING	14	Terminal paging (nonzero is on). This only works when PU$MODE is set to 1.
PU$PAGENUMBER	15	Returns the current page number.
PU$DISABLE	16	0 is returned if *print.channel* is enabled; and a 1 is returned if *print.channel* is disabled.
PU$CONNECT	17	Returns the number of a connected print channel or an empty string if no print channels are connected.
PU$LINESLEFT	1002	Lines left before new page needed. Returns erroneous values for the terminal if cursor addressing is used, if a line wider than the terminal is printed, or if terminal input has occurred.
PU$HEADERLINES	1003	Lines used by current header.
PU$FOOTERLINES	1004	Lines used by current footer.
PU$DATALINES	1005	Lines between current header and footer.
PU$DATACOLUMNS	1006	Columns between left margin and device width.

The PU$SPOOLFLAGS Key

The PU$SPOOLFLAGS key refers to a 32-bit option word that controls a number of print options. This is implemented as a 16-bit word and a 16-bit extension word. (Thus bit 21 refers to bit 5 of the extension word.) The bits are assigned as follows:

Bit	Description
1	Uses FORTRAN-format mode. This allows the attaching of vertical format information to each line of the data file. The first character position of each line from the file does not appear in the printed output, and is interpreted as follows:
	Character — **Meaning**
	0 — Advances two lines.
	1 — Ejects to the top of the next page.
	+ — Overprints the last line.
	Space — Advances one line.
	– — Advances three lines (skip two lines). Any other character is interpreted as advance one line.
3	Generates line numbers at the left margin.
4	Suppresses header page.
5	Suppresses final page eject after printing.
12	Spools the number of copies specified in an earlier !SETPU call.
21	Places the job in the spool queue in the hold state.
22	Retains jobs in the spool queue in the hold state after they have been printed.
other	All the remaining bits are reserved.

Equate Names for Return Code

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is ERRD.INS.IBAS. Use the

$INCLUDE statement to insert this file if you want to use equate names. The following list shows the codes returned in the argument *return.code*:

Code	Meaning
0	No error
E$BKEY	Bad key (*key* is out of range)
E$BPAR	Bad parameter (value of *new.value* is out of range)
E$BUNT	Bad unit number (value of *print.channel* is out of range)
E$NRIT	No write (attempt to set a read-only parameter)

Examples

In the following example, the file containing the parameter key equate names is inserted with the $INCLUDE compiler directive. Later, the top margin parameter for logical print channel 0 is interrogated. Print channel 0 is the terminal unless a prior PRINTER ON statement has been executed to direct output to the default printer. The top margin setting is returned in the argument TM.SETTING. Return codes are returned in the argument RETURN.CODE.

```
$INCLUDE SYSCOM GETPU.INS.IBAS
CALL !GETPU(PU$TOPMARGIN,0,TM.SETTING,RETURN.CODE)
```

The next example does the same as the previous example, but uses the key 4 instead of the equate name PU$TOPMARGIN. Because the key number is used, it is not necessary for the insert file GETPU.INS.IBAS to be included.

```
CALL !GETPU(4,0,TM.SETTING,RETURN.CODE)
```

The last example returns the current deferred time on print channel 0 in the variable TIME.RET:

```
CALL !GETPU(PU$DEFERTIME,0,TIME.RET,RETURN.CODE)
```

Syntax

CALL !GET.USER.COUNTS (*uv.users*, *max.uv.users*, *os.users*)

Description

Use the !GET.USER.COUNTS subroutine to return a count of UniVerse and system users. If any value cannot be retrieved, a value of –1 is returned.

uv.users (output) is the current number of UniVerse users.

max.uv.users (output) is the maximum number of licensed UniVerse users allowed on your system.

os.users (output) is the current number of operating system users.

!INLINE.PROMPTS subroutine

Syntax

CALL !INLINE.PROMPTS (*result*, *string*)

Description

Use the !INLINE.PROMPTS subroutine to evaluate a string that contains in-line prompts. In-line prompts have the following syntax:

<<{ *control*, }...*text*{ ,*option* }>>

result (output) is the variable that contains the result of the evaluation.

string (input) is the string containing an in-line prompt.

control specifies the characteristics of the prompt, and can be one of the following:

@(CLR)	Clears the terminal screen.
@(BELL)	Rings the terminal bell.
@(TOF)	Issues a formfeed character: in most circumstances this results in the cursor moving to the top left of the screen.
@(*col*,*row*)	Prompts at the specified column and row number on the terminal.
A	Always prompts when the in-line prompt containing the control option is evaluated. If you do not specify this option, the input value from a previous execution of the prompt is used.
C*n*	Specifies that the *n*th word on the command line is used as the input value. (Word 1 is the verb in the sentence.)
F(*filename*,*record*.*id*[,*fm*[,*vm*[,*sm*]]])	Takes the input value from the specified record in the specified file, and optionally, extracts a value (@VM), or subvalue (@SM), from the field (@FM). This option cannot be used with the file dictionary.
I*n*	Takes the *n*th word from the command line, but prompts if the word is not entered.
R(*string*)	Repeats the prompt until an empty string is entered. If *string* is specified, each response to the prompt is appended by *string* between each entry. If *string* is not specified, a space is used to separate the responses.

P	Saves the input from an in-line prompt. The input is then used for all in-line prompts with the same prompt text. This is done until the saved input is overwritten by a prompt with the same prompt text and with a control option of A, C, I, or S, or until control returns to the UniVerse prompt. The P option saves the input from an in-line prompt in the current paragraph, or in other paragraphs.
S*n*	Takes the *n*th word from the command (as in the I*n* control option), but uses the most recent command entered at the UniVerse system level to execute the paragraph, rather than an argument in the paragraph. This is useful where paragraphs are nested.
text	The prompt to be displayed.
option	A valid conversion code or pattern match. A valid conversion code is one that can be used with the ICONV function. Conversion codes must be enclosed in parentheses. A valid pattern match is one that can be used with the MATCHING keyword.

If the in-line prompt has a value, that value is substituted for the prompt. If the in-line prompt does not have a value, the prompt is displayed to request an input value when the function is executed. The value entered at the prompt is then substituted for the in-line prompt.

Note: Once a value has been entered for a particular prompt, the prompt continues to have that value until a !CLEAR.PROMPTS subroutine is called, or control option A is specified. A !CLEAR.PROMPTS subroutine clears all the values that have been entered for in-line prompts.

You can enclose prompts within prompts.

Example

```
A = ""
CALL !INLINE.PROMPTS(A,"You have requested the <<Filename>>
file")
PRINT "A"
```

The following output is displayed:

```
Filename=PERSONNEL
You have requested the PERSONNEL file
```

!INTS subroutine

Syntax

CALL !INTS (*result*, *dynamic.array*)

Description

Use the !INTS subroutine to retrieve the integer portion of elements in a dynamic array.

result (output) contains a dynamic array that comprises the integer portions of the elements of *dynamic.array*.

dynamic.array (input) is the dynamic array to process.

The !INTS subroutine returns a dynamic array, each element of which contains the integer portion of the numeric value in the corresponding element of the input *dynamic.array*.

Example

```
A=33.0009:@VM:999.999:@FM:-4.66:@FM:88.3874
CALL !INTS(RESULT,A)
```

The following output is displayed:

```
33VM999FM-4FM88
```

Syntax

CALL !MAKE.PATHNAME (*path1*, *path2*, *result*, *status*)

Description

Use the !MAKE.PATHNAME subroutine to construct the full pathname of a file. The !MAKE.PATHNAME subroutine can be used to:

- Concatenate two strings to form a pathname. The second string must be a relative path.
- Obtain the fully qualified pathname of a file. Where only one of *path1* or *path2* is given, !MAKE.PATHNAME returns the pathname in its fully qualified state. In this case, any filename you specify does not have to be an existing filename.
- Return the current working directory. To do this, specify both *path1* and *path2* as empty strings.

path1 (input) is a filename or partial pathname. If *path1* is an empty string, the current working directory is used.

path2 (input) is a relative pathname. If *path2* is an empty string, the current working directory is used.

result (output) is the resulting pathname.

status (output) is the returned status of the operation. 0 indicates success. Any other number indicates either of the following errors:

IE$NOTRELATIVE	*path*2 was not a relative pathname.
IE$PATHNOTFOUND	The pathname could not be found when !MAKE.PATHNAME tried to qualify it fully.

Example

In this example, the user's working directory is */usr/accounts*:

```
ENT = "ledger"
CALL !MAKE.PATHNAME(ENT,"",RESULT,STATUS)
IF STATUS = 0
THEN PRINT "Full name = ":RESULT
```

The following result is displayed:

```
Full name = /usr/accounts/ledger
```

!MATCHES subroutine

Syntax

CALL !MATCHES (*result*, *dynamic.array*, *match.pattern*)

Description

Use the !MATCHES subroutine to test whether each element of one dynamic array matches the patterns specified in the elements of the second dynamic array. Each element of *dynamic.array* is compared with the corresponding element of *match.pattern*. If the element in *dynamic.array* matches the pattern specified in *match.pattern*, 1 is returned in the corresponding element of *result*. If the element from *dynamic.array* is not matched by the specified pattern, 0 is returned.

result (output) is a dynamic array containing the result of the comparison on each element in *dynamic array1*.

dynamic.array (input) is the dynamic array to be tested.

match.pattern (input) is a dynamic array containing the match patterns.

When *dynamic.array* and *match.pattern* do not contain the same number of elements, the behavior of !MATCHES is as follows:

- *result* always contains the same number of elements as the longer of *dynamic.array* or *match.pattern*.
- If there are more elements in *dynamic.array* than in *match.pattern*, the missing elements are treated as though they contained a pattern that matched an empty string.
- If there are more elements in *match.pattern* than in *dynamic.array*, the missing elements are treated as though they contained an empty string.

Examples

The following example returns the value of the dynamic array as 1VM1VM1:

```
A='AAA4A4':@VM:2398:@VM:'TRAIN'
B='6X':@VM:'4N':@VM:'5A'
CALL !MATCHES(RESULT,A,B)
```

In the next example, there are missing elements in *match.pattern* that are treated as though they contain a pattern that matches an empty string. The result is 0**VM**0**SM**0**FM**1**FM**1.

```
R='AAA':@VM:222:@SM:'CCCC':@FM:33:@FM:'DDDDDD'
S='4A':@FM:'2N':@FM:'6X'
CALL !MATCHES(RESULT,R,S)
```

In the following example, the missing element in match.pattern is used as a test for an empty string in dynamic.array, and the result is 1**VM**1**FM**1:

```
X='AAA':@VM:@FM:''
Y='3A':@FM:'3A'
CALL !MATCHES(RESULT,X,Y)
```

!MESSAGE subroutine

Syntax

CALL !MESSAGE (*key, username, usernum, message, status*)

Description

Use the !MESSAGE subroutine to send a message to another user on the system. !MESSAGE lets you change and report on the current user's message status.

key (input) specifies the operation to be performed. You specify the option you require with the *key* argument, as follows:

IK$MSGACCEPT	Sets message status to accept.
IK$MSGREJECT	Sets message status to reject.
IK$MSGSEND	Sends message to user.
IK$MSGSENDNOW	Sends message to user now.
IK$MSGSTATUS	Displays message status of user.

username (input) is the name of the user, or the TTY name, for send or status operations.

usernum (input) is the number of the user for send / status operations.

message (input) is the message to be sent.

status (output) is the returned status of the operation as follows:

0	The operation was successful.
IE$NOSUPPORT	You specified an unsupported *key* option.
IE$KEY	You specified an invalid *key* option.
IE$PAR	The *username* or *message* you specified was not valid.
IE$UNKNOWN.USER	You tried to send a message to a user who is not logged in to the system.
IE$SEND.REQ.REC	The sender does not have the MESSAGERECEIVE option enabled.
IE$MSG.REJECTED	One or more users have the MESSAGEREJECT mode set.

Note: The value of *message* is ignored when *key* is set to IK$MSGACCEPT, IK$MSGREJECT, or IK$MSGSTATUS.

Example

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
IF CODE # 0
THEN CALL !REPORT.ERROR ('MY.COMMAND','!MESSAGE',CODE)
```

!REPORT.ERROR subroutine

Syntax

CALL !REPORT.ERROR (*command*, *subroutine*, *code*)

Description

Use the !REPORT.ERROR subroutine to print explanatory text for a UniVerse or operating system error code.

command is the name of the command that used the subroutine in which an error was reported.

subroutine is the name of the subroutine that returned the error code.

code is the error code.

The general format of the message printed by !REPORT.ERROR is as follows:

```
Error: Calling subroutine from command. system error code:
message.text.
```

system is the operating system, or UniVerse.

Text for values of *code* in the range 0 through 9999 is retrieved from the operating system. Text for values of *code* over 10,000 is retrieved from the SYS.MESSAGES file. If the code has no associated text, a message to that effect is displayed. Some UniVerse error messages allow text to be inserted in them. In this case, *code* can be a dynamic array of the error number, followed by one or more parameters to be inserted into the message text.

Examples

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
IF CODE # 0
THEN CALL !REPORT.ERROR ('MY.COMMAND','!MESSAGE',CODE)
```

If *code* was IE$SEND.REQ.REC, !REPORT.ERROR would display the following:

```
Error calling "!MESSAGE" from "MY.COMMAND" UniVerse error
1914: Warning: Sender requires "receive" enabled!
```

The next example shows an error message with additional text:

```
CALL !MESSAGE (KEY,USERNAME,USERNUMBER,MESSAGE,CODE)
IF CODE # 0
THEN CALL !REPORT.ERROR
('MY.COMMAND','!MESSAGE',CODE:@FM:USERNAME)
```

If *code* was IE$UNKNOWN.USER, and the user ID was joanna, !REPORT.ERROR would display the following:

```
Error calling "!MESSAGE" from "MY.COMMAND" UniVerse error
1757: joanna is not logged on
```

!SET.PTR subroutine

Syntax

CALL !SET.PTR (*print.channel, width, length, top.margin, bottom.margin, mode, options*)

Description

Use the !SET.PTR subroutine to set options for a logical print channel. This subroutine provides the same functionality as the UniVerse SETPTR command.

print.channel is the logical printer number, –1 through 255. The default is 0.

width is the page width. The default is 132.

length is the page length. The default is 66.

top.margin is the number of lines left at the top of the page. The default is 3.

bottom.margin is the number of lines left at the bottom of the page. The default is 3.

mode is a number 1 through 5 that indicates the output medium, as follows:

1 - Line Printer Spooler Output (default).

2, 4, 5 - Assigned Device. To send output to an assigned device, you must first assign the device to a logical print channel, using the UniVerse ASSIGN command. The ASSIGN command issues an automatic SETPTR command using the default parameters, except for mode, which it sets to 2. Use !SET.PTR only if you have to change the default parameters.

3 - Hold File Output. Mode 3 directs all printer output to a file called &HOLD&. If a &HOLD& file does not exist in your account, !SET.PTR creates the file and its dictionary (D_&HOLD&). You must execute !SET.PTR with mode 3 before each report to create unique report names in &HOLD&. If the report exists with the same name, the new report overwrites.

options are any of the printer options that are valid for the SETPTR command. These must be separated by commas, and enclosed by valid quotation marks.

If you want to leave a characteristic unchanged, supply an empty string argument and specify the option NODEFAULT. If you want the default to be selected, supply an empty string argument without specifying the NODEFAULT option.

Printing on the Last Line and Printing a Heading

If you print on the last line of the page or screen and use a HEADING statement to print a heading, your printout will have blank pages. The printer or terminal is set to advance to the top of the next page when the last line of the page or screen is printed. The HEADING statement is set to advance to the top of the next page to print the heading.

Example

The following example sets the options so that printing is deferred until 12:00, and the job is retained in the queue:

```
CALL !SET.PTR (0,80,60,3,3,1,'DEFER 12:00,RETAIN')
```

!SETPU subroutine

Syntax

CALL !SETPU (*key*, *print.channel*, *new.value*, *return.code*)

Description

Use the !SETPU subroutine to set individual parameters of any logical print channel.

Unlike !SET.PTR, you can only specify individual parameters to change; you do not need to specify parameters you do not want to change. See the description of the !GETPU subroutine for a way to read individual *print.channel* parameters.

key is a number indicating the parameter to be set (see "Equate Names for Keys").

print.channel is the logical print channel, designated by –1 through 255.

new.value is the value to which you want to set the parameter.

return.code is the returned error code (see "Equate Names for Return Code").

The !SETPU subroutine lets you change individual parameters of logical print channels as designated by *print.channel.* Print channel 0 is the terminal unless a PRINTER ON statement has been executed to send output to the default printer. If you specify print channel –1, the output is directed to the terminal, regardless of the status of PRINTER ON or OFF.

Equate Names for Keys

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is GETPU.INS.IBAS. Use the $INCLUDE compiler directive to insert this file if you want to use the equate names. For a description of the $INCLUDE compiler directive, see Chapter 3. The following list shows the equate names and keys for the parameters:

Mnemonic	Key	Parameter
PU$MODE	1	Printer mode.
PU$WIDTH	2	Device width (columns).
PU$LENGTH	3	Device length (lines).
PU$TOPMARGIN	4	Top margin (lines).
PU$BOTMARGIN	5	Bottom margin (lines).

Mnemonic	Key	Parameter
PU$SPOOLFLAGS	7	Spool option flags (see “The PU$SPOOLFLAGS Key”).
PU$DEFERTIME	8	Spool defer time. This cannot be 0.
PU$FORM	9	Spool form (string).
PU$BANNER	10	Spool banner or hold filename (string).
PU$LOCATION	11	Spool location (string).
PU$COPIES	12	Spool copies. A single copy can be returned as 1 or 0.
PU$PAGING	14	Terminal paging (nonzero is on). This only works when PU$MODE is set to 1.
PU$PAGENUMBER	15	Sets the next page number.

The PU$SPOOLFLAGS Key

The PU$SPOOLFLAGS key refers to a 32-bit option word that controls a number of print options. This is implemented as a 16-bit word and a 16-bit extension word. (Thus bit 21 refers to bit 5 of the extension word.) The bits are assigned as follows:

Bit	Description
1	Uses FORTRAN-format mode. This allows the attaching of vertical format information to each line of the data file. The first character position of each line from the file does not appear in the printed output, and is interpreted as follows:
	Character — **Meaning**
	0 — Advances two lines.
	1 — Ejects to the top of the next page.
	+ — Overprints the last line.
	Space — Advances one line.
	– — Advances three lines (skip two lines). Any other character is interpreted as advance one line.
3	Generates line numbers at the left margin.
4	Suppresses header page.
5	Suppresses final page eject after printing.

Bit	Description
12	Spools the number of copies specified in an earlier !SETPU call.
21	Places the job in the spool queue in the hold state.
22	Retains jobs in the spool queue in the hold state after they have been printed.
other	All the remaining bits are reserved.

Equate Names for Return Code

An insert file of equate names is provided to allow you to use mnemonics rather than key numbers. The name of the insert file is ERRD.INS.IBAS. Use the $INCLUDE statement to insert this file if you want to use equate names. The following list shows the codes returned in the argument *return.code*:

Code	Meaning
0	No error
E$BKEY	Bad key (*key* is out of range)
E$BPAR	Bad parameter (value of *new.value* is out of range)
E$BUNT	Bad unit number (value of *print.channel* is out of range)
E$NRIT	No write (attempt to set a read-only parameter)

Printing on the Last Line and Printing a Heading

If you print on the last line of the page or screen and use a HEADING statement to print a heading, your printout will have blank pages. The printer or terminal is set to advance to the top of the next page or screen when the last line of the page or screen is printed. The HEADING statement is set to advance to the top of the next page to print the heading.

Examples

In the following example, the file containing the parameter key equate names is inserted with the $INCLUDE compiler directive. Later, the top margin parameter for logical print channel 0 is set to 10 lines. Return codes are returned in the argument RETURN.CODE.

```
$INCLUDE SYSCOM GETPU.INS.IBAS
CALL !SETPU(PU$TOPMARGIN,0,10,RETURN.CODE)
```

The next example does the same as the previous example, but uses the key 4 instead of the equate name PU$TOPMARGIN. Because the key is used, it is not necessary for the insert file GETPU.INS.IBAS to be included.

```
CALL !SETPU(4,0,10,RETURN.CODE)
```

!TIMDAT subroutine

Syntax

CALL !TIMDAT (*variable*)

Description

Use the !TIMDAT subroutine to return a dynamic array containing the time, date, and other related information. The !TIMDAT subroutine returns a 13-element dynamic array containing information shown in the following list.

variable is the name of the variable to which the dynamic array is to be assigned.

Field	Description
1	Month (two digits)
2	Day of month (two digits)
3	Year (two digits)
4	Minutes since midnight (integer)
5	Seconds into the minute (integer)
6	Ticks[1] of last second since midnight (integer). Always returns 0.
7	CPU seconds used since entering UniVerse.
8	Ticks of last second used since login (integer).
9	Disk I/O seconds used since entering UniVerse. Always returns –1.
10	Ticks of last disk I/O second used since login (integer). Always returns –1.
11	Number of ticks per second
12	User number
13	Login ID (user ID)

1. Tick refers to the unit of time your system uses to measure real time.

Use the following functions for alternative ways of obtaining time and date information:

Use this function…	To obtain this data…
DATE ( )	Data in fields 1, 2, and 3 of the dynamic array returned by the !TIMDAT subroutine
TIME ( )	Data in fields 4, 5, and 6 of the dynamic array returned by the !TIMDAT subroutine
@USERNO	User number
@LOGNAME	Login ID (user ID)

Example

```
CALL !TIMDAT(DYNARRAY)
FOR X = 1 TO 13
   PRINT 'ELEMENT ':X:', DYNARRAY
NEXT X
```

!USER.TYPE subroutine

Syntax

CALL !USER.TYPE (*type*, *admin*)

Description

Use the !USER.TYPE subroutine to return the user type of the current process and a flag to indicate if the user is a UniVerse Administrator.

type is a value that indicates the type of process making the subroutine call. *type* can be either of the following:

Equate Name	Value	Meaning
U$NORM	1	Normal user
U$PH	65	Phantom

admin is a value that indicates if the user making the call is a UniVerse administrator. Possible values of *admin* are 1, if the user is a UniVerse administrator, and 0, if the user is not a UniVerse administrator.

An insert file of equate names is provided for the !USER.TYPE values. To use the equate names, specify the directive $INCLUDE SYSCOM USER_TYPES.H when you compile your program. (For PI/open compatibility you can specify $INCLUDE SYSCOM USER_TYPES.INS.IBAS.)

Example

In the following example, the !USER.TYPE subroutine is called to determine the type of user. If the user is a phantom, the program stops. If the user is not a phantom, the program sends a message to the terminal and continues processing.

```
ERROR.ACCOUNTS.FILE: CALL !USER.TYPE(TYPE, ADMIN)
IF TYPE = U&PH THEN STOP
  ELSE PRINT 'Error on opening ACCOUNTS file'
```

Syntax

CALL !VOC.PATHNAME (*data/dict*, *voc.entry*, *result*, *status*)

Description

Use the !VOC.PATHNAME subroutine to extract the pathnames for the data file or the file dictionary of a specified VOC entry.

data/dict (input) indicates the file dictionary or data file, as follows:

IK$DICT or 'DICT' returns the pathname of the file dictionary of the specified VOC entry.

IK$DATA or '' returns the pathname (or pathnames for distributed files), of the data file of the specified VOC entry.

voc.entry is the record ID in the VOC.

result (output) is the resulting pathnames.

status (output) is the returned status of the operation.

An insert file of equate names is provided for the *data/dict* values. To use the equate names, specify the directive $INCLUDE SYSCOM INFO_KEYS.H when you compile your program. (For PI/open compatibility you can specify $INCLUDE SYSCOM INFO_KEYS.INS.IBAS.)

The result of the operation is returned in the *status* argument, and has one of the following values:

Value	Result
0	The operation executed successfully.
IE$PAR	A bad parameter was used in *data/dict* or *voc.entry*.
IE$RNF	The VOC entry record cannot be found.

Example

```
CALL !VOC.PATHNAME (IK$DATA,"VOC",VOC.PATH,STATUS)
IF STATUS = 0
THEN PRINT "VOC PATHNAME = ":VOC.PATH
```

If the user's current working directory is */usr/account*, the output is:

```
VOC PATHNAME =  /usr/accounts/VOC
```

Index

Symbols

– operator 2-11
! statement 1-4
!ASYNC subroutine F-4
!ERRNO subroutine F-6
!FCMP subroutine F-7
!GET.PARTNUM subroutine F-8
!GET.PATHNAME subroutine F-10
!GET.USER.COUNTS subroutine F-15
!GETPU subroutine F-11
!INLINE.PROMPTS subroutine F-16
!INTS subroutine F-18
!MAKE.PATHNAME subroutine F-19
!MATCHES subroutine F-20
!MESSAGE subroutine F-22
!REPORT.ERROR subroutine F-24
!SET.PTR subroutine F-26
!SETPU subroutine F-28
!TIMDAT subroutine F-32
!USER.TYPE subroutine F-34
!VOC.PATHNAME subroutine F-35
operator 2-15
#INCLUDE statement 3-4
$CHAIN statement 3-4
$DEFINE statement 3-5
$INCLUDE statement 3-4
$INSERT 3-2
$INSERT statement 3-4, 3-5
$OPTIONS statement 3-6
    STATIC.DIM option 2-6
    VEC.MATH option 2-12, 2-19
$UNDEFINE statement 3-5
&PH& file 3-1
* operator 2-11
* statement 1-4
** operator 2-11
+ operator 2-11
+= operator 2-18
/ operator 2-11
: operator 2-12
:= operator 2-18
< operator 2-15
<= operator 2-15
<> operator 2-15
–= operator 2-18
= operator 2-15, 2-18
> operator 2-15
>< operator 2-15
>= operator 2-15
@ABORT.CODE variable E-1
@ACCOUNT variable E-1
@AM variable E-1
@ANS variable E-1
@AUTHORIZATION variable E-1
@COMMAND variable E-1
@COMMAND.STACK variable E-2
@CONV variable E-2
@CRTHIGH variable E-1
@CRTWIDE variable E-1
@DATA.PENDING variable E-2
@DATE variable E-2
@DAY variable E-2
@DICT variable E-2
@FILE.NAME variable E-2
@FILENAME variable E-2
@FM variable E-2
@FORMAT variable E-2
@HDBC variable E-2
@HEADER variable E-2
@HENV variable E-2
@HSTMT variable E-2

@ID variable E-2
@IM variable E-2
@ISOLATION variable 4-10, E-2
@LEVEL variable E-2
@LOGNAME variable E-2
@LPTRHIGH variable E-2
@LPTRWIDE variable E-3
@MONTH variable E-3
@MV variable E-3
@NB variable E-3
@ND variable E-3
@NI variable E-3
@NS variable E-3
@NULL variable 2-3, 2-13, E-3
@NULL.STR variable 2-4, 2-13, E-3
@OPTION variable E-3
@PARASENTENCE variable E-3
@PATH variable E-3
@RECCOUNT variable E-3
@RECORD variable E-3
@SCHEMA variable E-4
@SELECTED variable E-4
@SENTENCE variable E-4
@SM variable E-4
@SQL.CODE variable E-4
@SQL.ERROR variable E-4
@SQL.STATE variable E-4
@SQL.WARNING variable E-4
@SQLPROC.NAME variable E-4
@SQLPROC.TX.LEVEL variable E-4
@STDFIL variable E-4
@SVM variable E-4
@SYS.BELL variable E-4
@SYSTEM.RETURN.CODE variable E-4
@SYSTEM.SET variable E-4
@TERM.TYPE variable E-4
@TIME variable E-4
@TM variable E-4
@TRANSACTION variable 4-10, E-5
@TRANSACTION.ID variable 4-10, E-5
@TRANSACTION.LEVEL variable 4-10, E-5
@TRUE variable E-5
@TTY variable E-5
@USER.NO variable E-5
@USER.RETURN.CODE variable E-5
@USER0 variable E-5
@USER1 variable E-5
@USER2 variable E-5
@USER3 variable E-5
@USER4 variable E-5
@USERNO variable E-5
@VM variable E-5
@WHO variable E-5
@YEAR variable E-5
^ operator 2-11

A

A conversion C-3
ACID properties 4-7
algebraic functions C-3
alphabetic characters 1-6
AND operator 2-17
arguments
  passing to subroutines 1-3
arithmetic operators 2-11–2-12
  and dynamic arrays 2-19
  and multivalued data 2-12
  and the null value 2-11
array variables 2-5–2-9
arrays
  dimensioned 2-5–2-6
  dynamic 2-6–2-9
  matrices 2-5
  standard 2-5, 2-6
  vectors 2-5
ASCII
  characters 2-2

CHAR(0) 2-2
CHAR(10) 2-2
CHAR(128) 2-3
codes B-1
conversion C-27
strings 1-2, 2-15
assignment operators 2-18
and substrings 2-14
assignment statements 1-4
atomicity property 4-7

B

BASIC character set
alphabetic 1-6
numeric 1-6
special 1-6
BASIC command 3-1–3-4
options 3-2–3-4
BASIC compiler
keywords 1-2
variables 1-2
BASIC language
application 1-1
extensions of 1-1
BASIC programs
cataloging 3-11–3-14
compiling 3-1–3-10
conditional compiling 3-6
definition 1-2
editing 1-7
flavors compatibility 3-6
global cataloging 3-12
listing 3-2
local cataloging 3-12
normal cataloging 3-12
object code 3-10
listing 5-12
printing 3-2, 3-4
running 3-11
storing 1-7
BASIC statements 1-2
binary conversion C-26
blank spaces
in source code 1-5
Break key 5-3

C

C conversion C-7
CALL statement 1-3
CAT operator 2-12
CATALOG command 3-13
catalog shared memory 3-14
cataloging BASIC programs
globally 3-12
locally 3-12
normally 3-12
CHAR(0) 2-2
CHAR(10) 2-2
CHAR(252) 2-6
CHAR(253) 2-6
CHAR(254) 2-6
character conversion C-18
character set conversion C-12
character strings 2-1–2-2
ASCII 2-2
constants 2-1
empty 2-3, 2-4
numeric 2-12
substrings 2-13
commas
in numeric constants 2-2
comments 1-4
common variables 2-5
compiler directives 3-4–3-8
concatenation
and the null value 2-13
conversion C-7
operator 2-12
conditional compiling 3-6
configurable parameters
ISOMODE 4-13
MAXRLOCK 4-15
TXMODE 4-9
consistency property 4-7

constants
character string 2-1
definition 2-4
fixed-point numeric 2-3
floating-point 2-3
numeric 2-2
control statements, *see* statements
conversion codes C-1–C-37
A C-3
C C-7
D C-8
F C-13
G C-16
L C-17
MB C-26
MC C-18
MD C-19
ML C-22
MO C-26
MP C-24
MR C-22
MX C-26
NR C-28
P C-29
Q C-30
R C-32
S (Soundex) C-33
S (substitution) C-34
T C-25, C-35
T*file* C-36

D

D conversion C-8
data
anomalies 4-12
character string 2-1–2-2
null value 2-3
numeric 2-2
preventing loss of 4-1
data types 2-1–2-4
logical 2-17
null value 2-3
date conversion C-8
date format
default C-8
deadlocks 4-5
DEBUG statement 5-2
debugger 5-1
DECATALOG command 3-14
decimal equivalents B-5
DELETE.CATALOG command 3-13, 3-14
delimiters
system 2-6
dimensioned arrays 2-5–2-6
dirty reads 4-12
durability property 4-8
dynamic arrays 2-6–2-9
and arithmetic operators 2-19
and null value 2-24
and operators 2-19–2-25
and the REUSE function 2-24
creating 2-8

E

ECS conversion C-12
edit editor 1-7
editor
edit 1-7
vi 1-7
empty string 2-3, 2-4
and pattern matching 2-16
EQ operator 2-15
error messages 3-10
exclusive file locks 4-5
expressions 2-9
format 2-9

F

F conversion C-13
field marks 2-6

file translation C-36
file variables 2-9
files
  &PH& 3-1
  type 1 1-7, 3-1
  type 19 1-7, 3-1
fixed-point constants 2-3
flavors
  compatibility 3-6
floating-point
  constants 2-3
  numbers 2-2
format expressions 2-9
formatting numbers C-22
functions
  intrinsic 1-2
  numeric 1-2
  string 1-2
  vector 2-19

G

G conversion C-16
GE operator 2-15
global cataloging 3-12
GOSUB statement 1-3
granularity 4-1
group extraction C-16
GT operator 2-15

H

hexadecimal
  conversion C-26
  equivalents B-5

I

ICONV function C-1
ICONVS function C-5
identifiers
  removing 3-5
  replacing 3-5
IF operator 2-16
INCLUDE statement 3-4
intent file locks 4-4
intrinsic functions, *see* functions
ISNULL function 2-3, 2-16
ISNULLS function 2-3, 2-16
isolation levels 4-14
  minimum locks for 4-14
  types 4-12
isolation property 4-8
ISOMODE parameter 4-13

K

keywords 1-2
  definition 1-3

L

L conversion C-17
labels
  statement 1-5
LE operator 2-15
length function C-17
line number table
  suppressing 3-2, 3-4
list variables, *see* select list variables
local cataloging 3-12
lock escalation
  example 4-15
locks
  compatibility 4-1
  deadlocks 4-5
  exclusive file 4-5
  granularity 4-1
  intent file 4-4
  shared file 4-4
  shared record
    READL locks, *see* shared record locks
  transactions and 4-8

types 4-2
update record
READU locks, *see* update record locks
well-formed writes and 4-13
logical operators 2-17
AND 2-17
and the null value 2-17
NOT 2-17
OR 2-17
lost updates 4-12
LT operator 2-15

M

masked character conversions C-18
MATCH operator 2-16
mathematical functions C-3, C-13
matrices 2-5
zero element 2-6
MAXRLOCK parameter 4-15
MB conversion C-26
MC conversion C-18
MD conversion C-19
messages
error 3-10
warning 3-10
ML conversion C-22
MO conversion C-26
MP conversion C-24
MR conversion C-22
multivalues and arithmetic operators 2-12
MX conversion C-26
MY conversion C-27

N

named common variables 2-5
names of variables 1-2
NE operator 2-15
nested transactions 4-6
example 4-9
properties 4-7
newline 1-5
nonrepeatable reads 4-12
normal cataloging 3-12
NOT operator 2-17
NR conversion C-28
null values
and concatenation operators 2-13
and dynamic arrays 2-24
and logical operators 2-17
definition 2-3
in arithmetic expressions 2-11
stored representation 2-4
numbers
floating-point 2-2
numeric
character strings 2-12
characters 1-6
constants 2-2
data 2-2
functions 1-2

O

object code 1-2, 3-10
listing 5-12
OCONV function C-1
OCONVS function C-5
octal conversion C-26
operators 2-10–2-25
and dynamic arrays 2-19–2-25
arithmetic 2-11–2-12
assignment 2-14, 2-18
concatenation 2-12
logical 2-17
pattern matching 2-16
relational 2-14–2-16
string 2-12
substring 2-13
OR operator 2-17

P

P conversion C-29
packed decimal conversion C-24
pattern matching 2-16, C-29
  and the empty string 2-16
PHANTOM command 3-1
phantom writes 4-12

Q

Q conversion C-30
quotation marks
  in character strings 2-2

R

R conversion C-32
RAID (debugger)
  commands 5-4
  description 5-1
  table of commands 5-4
RAID command 5-2
  options 5-2
  suppressing execution of 3-2
range function C-32
relational operators 2-14–2-16
REM statement 1-4
RETURN statement 1-3
REUSE function 2-24
Roman numeral conversion C-28
RUN command 3-11
  options 3-11

S

S (Soundex) conversion C-33
S (substitution) conversion C-34
scientific notation 2-3
select list variables 2-9
sequential file I/O 1-5
serializability
  lost updates and 4-13
  property 4-8
SET TRANSACTION ISOLATION LEVEL statement 4-9
SETPTR command F-26
shared file locks 4-4
shared memory 3-14
shared record locks
SOUNDEX conversion C-33
source code 1-2
  comments 1-4
  spaces in 1-5
  syntax 1-3
  tabs in 1-5
spaces
  in numeric constants 2-2
  in source code 1-5
special characters 1-6
standard arrays 2-6
  matrices 2-5
  vectors 2-5
statement labels
  cross-reference table of 3-2
  definition 1-5
statements 1-2
  assignment 1-4
  control 1-4
  types 1-4
STATIC.DIM option 2-6
string
  functions 1-2, C-13
  operators 2-12
strings
  comparing 2-14
  see also character strings
SUBROUTINE statement 1-3
subroutines
  definition 1-3
  list F-2
  passing arguments to 1-3
  vector functions 2-19
substitution C-34

substring operator 2-13
substrings 2-13–2-14
    and the assignment operator 2-14
    definition 2-13
    extracting 2-14
subtransactions 4-6
subvalue marks 2-6
symbol table
    suppressing 3-2, 3-4
syntax
    notation viii
    source code 1-3
system delimiters 2-6

T

T conversion C-25, C-35
tabs
    in source code 1-5
text extraction C-35
T*file* conversion C-36
time conversion C-25
transaction variables
    @ISOLATION 4-10
    @TRANSACTION variable 4-10
    @TRANSACTION.ID
        variable 4-10
    @TRANSACTION.LEVEL
        variable 4-10
transactions
    active 4-6
    child 4-6
    example 4-9
    isolation levels 4-9
    locks and 4-8
    nested 4-6
    parent 4-6
    properties 4-7
    restrictions 4-10
    subtransactions 4-6
    top-level 4-6
TXMODE parameter 4-9
type 1 files 1-7, 3-1
type 19 files 1-7, 3-1

U

unnamed common variables 2-5
update record locks
UV/Net
    writing to remote files 4-10

V

value marks 2-6
variables 1-2, 2-4–2-9
    array 2-5–2-9
    definition 1-2
    file 2-9
    in RAID 5-3
    named common 2-5
    names 1-2, 2-5
    select list 2-9
    transaction 4-10
    unnamed common 2-5
VEC.MATH option 2-12, 2-19
vector functions 2-19
    as subroutines 2-19
vectors 2-5
    zero element 2-6
vi editor 1-7
VLIST command 5-12
    suppressing execution of 3-2

W

warning messages 3-10
well-formed write 4-13

Z

zero element 2-6

To help us provide you with the best documentation possible, please make your comments and suggestions concerning this manual on this form and fax it to us at **508-366-3669, attention Technical Publications Manager**. All comments and suggestions become the property of VMARK SOFTWARE, INC. We greatly appreciate your comments.

Comments

Name: _____ Date: _____

Position: _____ Dept: _____

Organization: _____ Phone: _____

Address: _____

_____

Name of Manual: **UniVerse BASIC Programmer's Guide**

Part Number: **70-9025-931**